Grant Number NAG8-093

A DIRECT-EXECUTION PARALLEL ARCHITECTURE FOR THE ADVANCED CONTINUOUS SIMULATION LANGUAGE (ACSL)

by

Chester C. Carroll
Cudworth Professor of Computer Architecture
Department of Electrical Engineering
College of Engineering
The University of Alabama
Tuscaloosa, Alabama

and

Jeffrey E. Owen
Graduate Research Assistant

Prepared for

National Aeronautics and Space Administration

Bureau of Engineering Research
The University of Alabama
May 1988

BER Report No. 424-17

The University of Alabama
College of Engineering
Bureau of Engineering Research
P.O. Box 1968
University, Alabama 35486
Telephone: (205) 348-1591

THE UNIVERSITY OF ALABAMA
COLLEGE OF ENGINEERING

The College of Engineering at The University of Alabama has an undergraduate enrollment of more than 2,300 students and a graduate enrollment exceeding 180. There are approximately 100 faculty members, a significant number of whom conduct research in addition to teaching.

Research is an integral part of the educational program, and research interests of the faculty parallel academic specialities. A wide variety of projects are included in the overall research effort of the College, and these projects form a solid base for the graduate program which offers fourteen different master's and five different doctor of philosophy degrees.

Other organizations on the University campus that contribute to particular research needs of the College of Engineering are the Charles L. Seebeck Computer Center, Geological Survey of Alabama, Marine Environmental Sciences Consortium, Mineral Resources Institute—State Mine Experiment Station, Mineral Resources Research Institute, Natural Resources Center, School of Mines and Energy Development, Tuscaloosa Metallurgy Research Center of the U.S. Bureau of Mines, and the Research Grants Committee.

This University community provides opportunities for interdisciplinary work in pursuit of the basic goals of teaching, research, and public service.

BUREAU OF ENGINEERING RESEARCH

The Bureau of Engineering Research (BER) is an integral part of the College of Engineering of The University of Alabama. The primary functions of the BER include: 1) identifying sources of funds and other outside support bases to encourage and promote the research and educational activities within the College of Engineering; 2) organizing and promoting the research interests and accomplishments of the engineering faculty and students; 3) assisting in the preparation, coordination, and execution of proposals, including research, equipment, and instructional proposals; 4) providing engineering faculty, students, and staff with services such as graphics and audiovisual support and typing and editing of proposals and scholarly works; 5) promoting faculty and staff development through travel and seed project support, incentive stipends, and publicity related to engineering faculty, students, and programs; 6) developing innovative methods by which the College of Engineering can increase its effectiveness in providing high quality educational opportunities for those with whom it has contact; and 7) providing a source of timely and accurate data that reflect the variety and depth of contributions made by the faculty, students, and staff of the College of Engineering to the overall success of the University in meeting its mission.

Through these activities, the BER serves as a unit dedicated to assisting the College of Engineering faculty by providing significant and quality service activities.

Grant Number NAG8-093

A DIRECT-EXECUTION PARALLEL ARCHITECTURE FOR THE ADVANCED CONTINUOUS SIMULATION LANGUAGE (ACSL)

by

Chester C. Carroll
Cudworth Professor of Computer Architecture

and

Jeffrey E. Owen
Graduate Research Assistant

Prepared for

The National Aeronautics and Space Administration

Bureau of Engineering Research
The University of Alabama
May 1988

BER Report No. 424-17

LIST OF ABBREVIATIONS

ACSL	Advanced Continuous Simulation Language
AMD	Advanced Micro Devices
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
EPROM	Erasable Programmable Read Only Memory
FPU	Floating Point Unit
HLL	High Level Language
I/O	Input/Output
MIPS	Million Instructions Per Second
PE	Processing Element
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
TI	Texas Instruments

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A direct-execution parallel architecture for the Advanced Continuous Simulation Language (ACSL) is presented which overcomes the traditional disadvantages encountered when simulations are executed on a digital computer. The incorporation of parallel processing allows the mapping of simulations onto a digital computer to be done in the same inherently parallel manner as they are currently mapped onto an analog computer. The direct-execution format maximizes the efficiency of the executed code since the need for a high level language compiler is eliminated. Resolution is greatly increased over that which is available with an analog computer without the sacrifice in execution speed normally expected with digital computer simulations.

Although this report covers all aspects of the new architecture, key emphasis is placed on the processing element configuration and the microprogramming of the ACSL constructs. The execution times for all ACSL constructs are computed using a model of a processing element based on the AMD 29000 CPU and the AMD 29027 FPU. The increase in execution speed provided by parallel processing is exemplified by comparing the derived execution times of two ACSL programs with the execution times for the same programs executed on a similar sequential architecture.

CHAPTER 1

INTRODUCTION

Analog computers have traditionally been chosen over digital computers for the simulation of physical systems because analog computer architectures are tailor-made for solving systems of simultaneous differential equations in an inherently parallel fashion. The main drawback of an analog computer is the low resolution of its outputs which will degrade the accuracy of the simulation. The traditional Von Neumann digital computer is capable of higher resolution, but it requires more computational time due to the sequential nature of its architecture. This report will present a digital computer architecture that overcomes the slow execution problem of a Von Neumann machine while providing greater resolution than normally possible with an analog computer.

This paper will examine a specific simulation language, ACSL, and use two techniques to improve its execution speed in order to simulate systems in real-time. These two techniques are the use of a direct-execution architecture to bypass the compiler, thereby increasing system efficiency and speed, and the incorporation of parallel processing in the system architecture to further maximize execution speed.

All aspects of the architecture will be examined, including the microprogramming of the ACSL constructs, the processing element configuration, the interconnection network, the I/O processor, and the functions performed by the allocater. From this analysis execution

times of two example ACSL programs will be derived in terms of the minimum real-time calculation interval.

With the addition of appropriate sensors and actuators, this architecture could be used to simulate physical systems while they interact with other physical systems in real-time. Assuming the modeling equations are accurate, the results of the simulation will be precisely the same as if the actual component had been used. Furthermore, if the system being simulated is a type of computer controlled system, then the same architecture used to model it could also be used to implement the component being simulated.

The Advanced Continuous Simulation Language, ACSL

The Advanced Continuous Simulation Language (ACSL) is used to model dynamic systems by time dependent, non-linear differential equations and/or transfer functions (Mitchell and Gauthier, Associates 1986). Simulation of physical systems is a standard and useful analysis tool used to test the design of a system prior to the actual construction of the proposed system. For example, a program written in ACSL to determine whether or not a pilot ejecting from his aircraft will strike the plane's vertical stablilizer is a much better approach than actually ejecting a test pilot to see if he clears the tail fin of the aircraft.

A Direct-Execution Architecture

There are several ways high-level languages can be implemented. Some architectures concentrate on hardware, some on software, and still others on implementation technology. In general, computer architectures to implement high-level languages fall into one of the classifications shown in the tree diagram in figure 1 (Milutinovic 1988). Indirect-

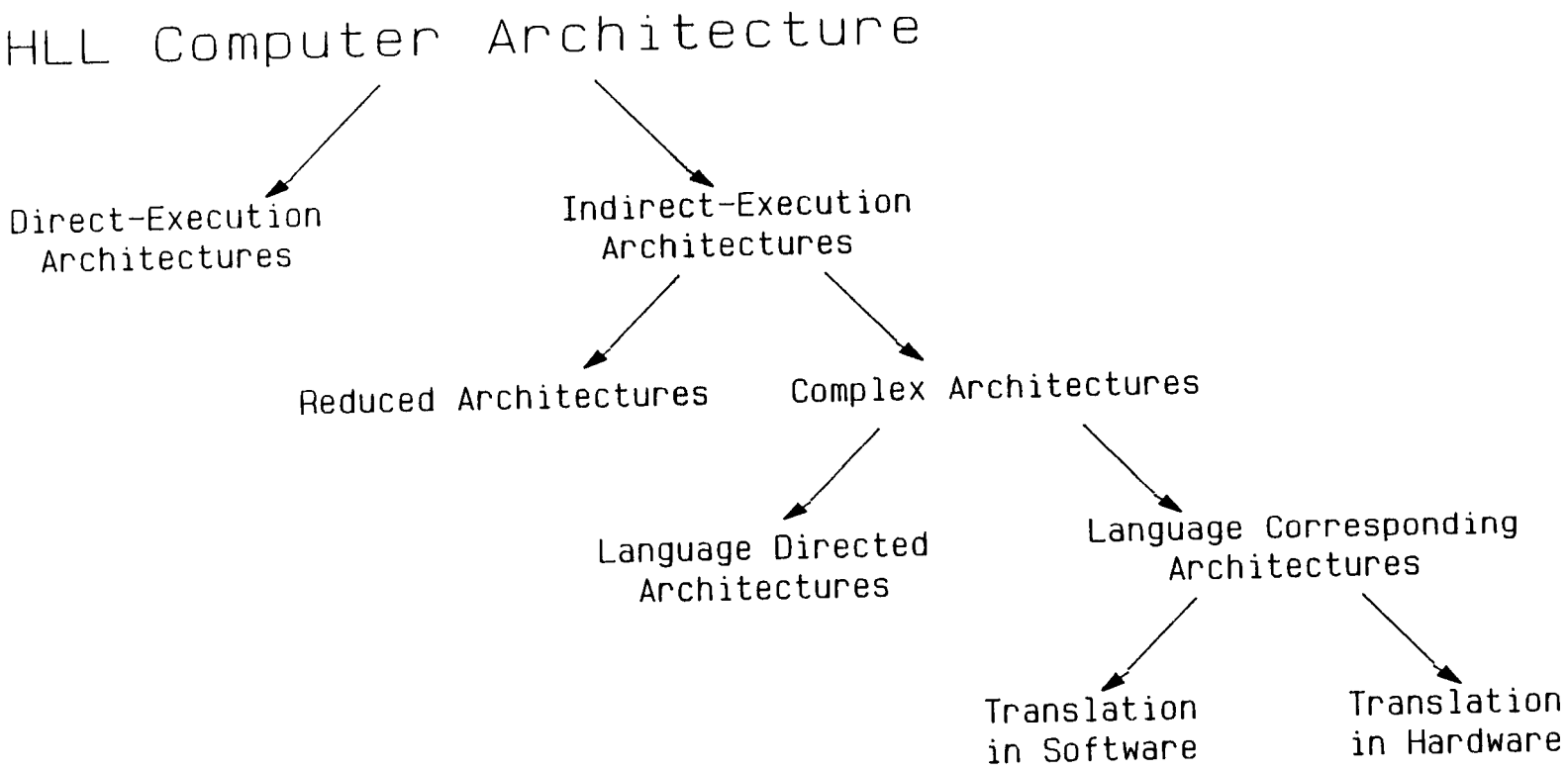


Figure 1. HLL Computer Architecture Classifications

execution architectures use software or hardware to translate or compile the high-level language program in to a form suitable for machine execution. Direct-execution architectures bypass the translation step by incorporating hardware or software functions to execute high-level language programs directly.

When a compiler is used to convert a high-level language to machine code, inefficiencies are introduced into the newly created machine code. These inefficiencies cause the system to operate below the maximum possible execution speed and cause the system to utilize more memory than would be required if the high-level language constructs were programmed in a more efficient manner. A direct-execution architecture can help solve these problems since each processing element is micro-programmed to execute HLL constructs directly, thereby eliminating the need for a compiler and the inefficiencies associated with it.

Parallel Processing

Parallel processing will be incorporated in the new architecture to increase execution speed. There is always a need in industry for faster execution speeds when modeling dynamic systems. The sequential machines used today simply cannot perform complex high-speed simulations in a real-time environment. With the introduction of parallel processing into a simulation language architecture, the simulation speeds for complex tasks will increase greatly over currently available simulation speeds. This has already been demonstrated at The University of California at Berkeley where the Department of Electrical Engineering and Computer Science is working on the Msplice parallel simulator for analog circuits. For some circuits a 32 processor version of Msplice runs as much as 25 times faster than a uniprocessor version (Howe 1987).

Parallel processing principles apply to any type device technology, so if it is stated that a parallel processing architecture is not needed because a higher speed technology (such as optical computers) will soon be available, please note that parallel processing can increase the speed of these devices in the same manner it is used to increase the performance of silicon devices.

How to Improve the Current ACSL Computer Design

To compile an ACSL program today, one first converts the ACSL source code to FORTRAN with a FORTRAN translator. The FORTRAN code is then compiled to create executable machine code. Each step taken to compile an ACSL program introduces inefficiencies into the resulting machine code. This process is illustrated in figure 2. Another problem with the current method is the fact that the vast majority of variables used in FORTRAN reside in main memory, thus making FORTRAN a memory intensive language. Even if the memory variables are residing in a data cache, execution would proceed at a higher rate if the variables were contained in internal CPU registers rather than in memory locations.

The direct-execution process rids an architecture of the problems stated above. The need for a FORTRAN translator and compiler is eliminated since the ACSL constructs are microprogrammed at each processing element (PE). This approach will result in more efficient code than could be achieved with a compiler. The memory access problem will be reduced by selecting a microprocessor with a large internal register file permitting program variables to reside in internal CPU registers.

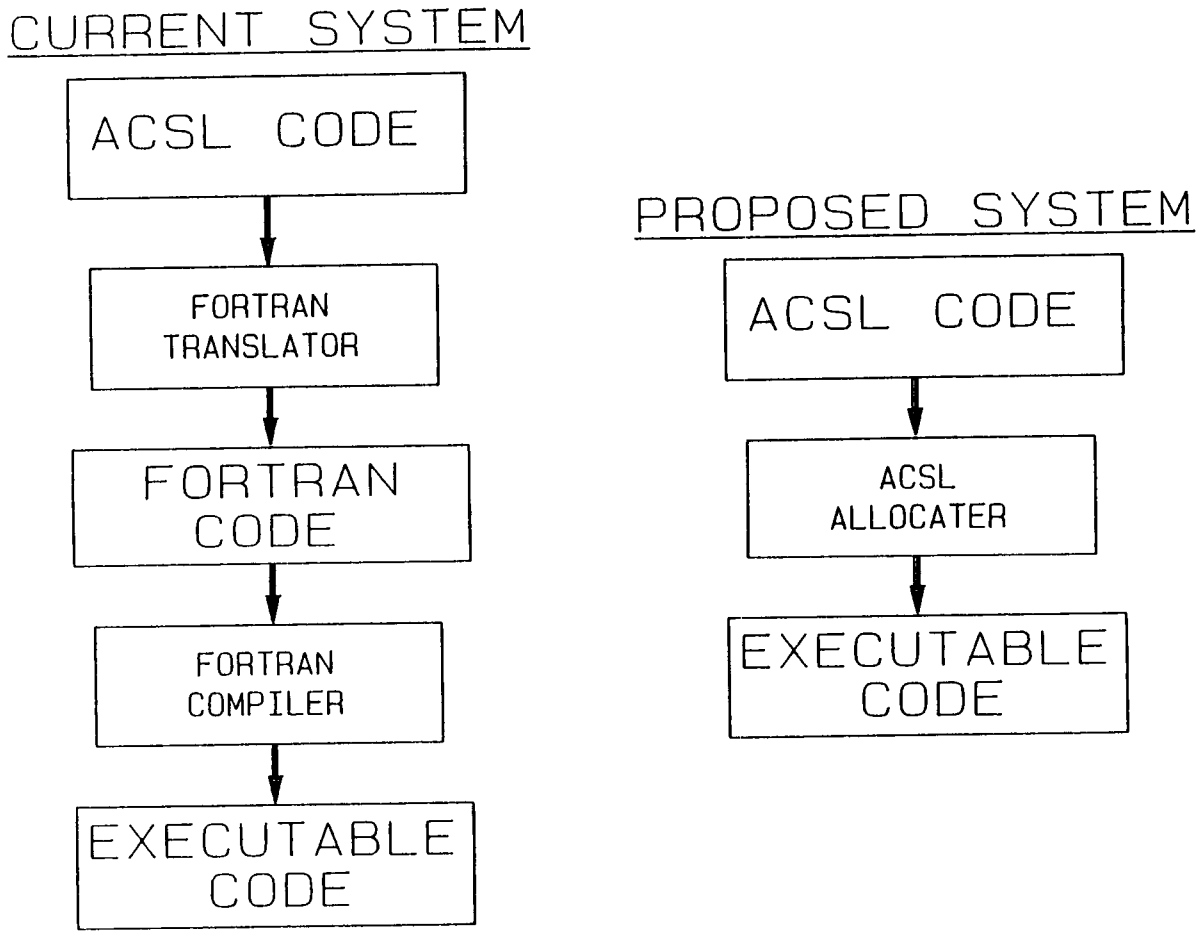


Figure 2. Current System versus Proposed System

CHAPTER 2

PARALLEL PROCESSING DESIGN CONSIDERATIONS

When designing a parallel processing architecture, there are several decisions to be made that are not considered when designing a typical sequential computing system. These decisions include the choice between a fine or course grained system, the method employed to organize memory, and what type interconnection network to use. These choices can either make an architecture fast and efficient, or they can bog down an architecture with inefficiencies to the point that a single high-speed processor can out-perform the parallel processing system.

Fine-grained or Course-grained Architecture

The granularity of an architecture describes the complexity of the functions that each PE performs. A fine-grained system's PE would perform simple functions such as an addition or multiplication. Conversely, a course-grained system's PE would be capable of more complex tasks, such as the evaluation of an entire equation with multiplications, additions, subtractions, divisions, etc. Granularity also expresses the ratio between computation and communication in a parallel program (Howe 1987). Fine-grained systems are typically characterized as having more communication overhead than course-grained systems.

The system under consideration will be implementing high-level language constructs, some of which are fairly complex; therefore, a course grained architecture will be employed in order to keep a moder-

ately complex construct microprogram executable within a single PE. Doing so will minimize communication requirements between PEs and decrease possible communication bottlenecks.

Shared Memory or Private Memory

In a shared memory system, multiple processors are connected to multiple memory banks through one or more buses. All memory in the system is contained in every processors memory map making all memory equally accessible by every processor. To access a memory location, the processor simply initiates a memory read or write cycle to the desired memory location. If no contention is present from the other processors, the requesting processor is allowed to access memory. This method provides the highest memory bandwidth but creates bottlenecks when several processors need access to the same memory bank at once.

In a private memory system, variables are passed to and from processors by way of a message passing scheme. To read a variable from another processor, the requesting processor sends a message to the processor holding the desired variable, and that processor sends a message back containing the variable. In general, private memory systems are usually efficient when the interactions between tasks are minimal, but shared memory systems can tolerate a higher degree of interaction between tasks without significant deterioration in performance (Howe 1987). With this in mind, if an architecture has a high degree of communication between tasks, a shared memory approach would be more efficient; but if the tasks had a low degree of inter-communication, a private memory approach would be better.

If a construct microprogram is considered to be a task in this ACSL architecture, there is then only a moderate amount of communication

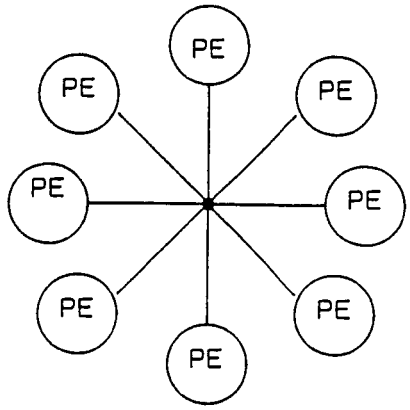
between tasks. This is primarily due to the fact that very few of the ACSL constructs contain significant amounts of parallelism; therefore, it appears that a private memory architecture would be the most efficient.

Interconnection Network

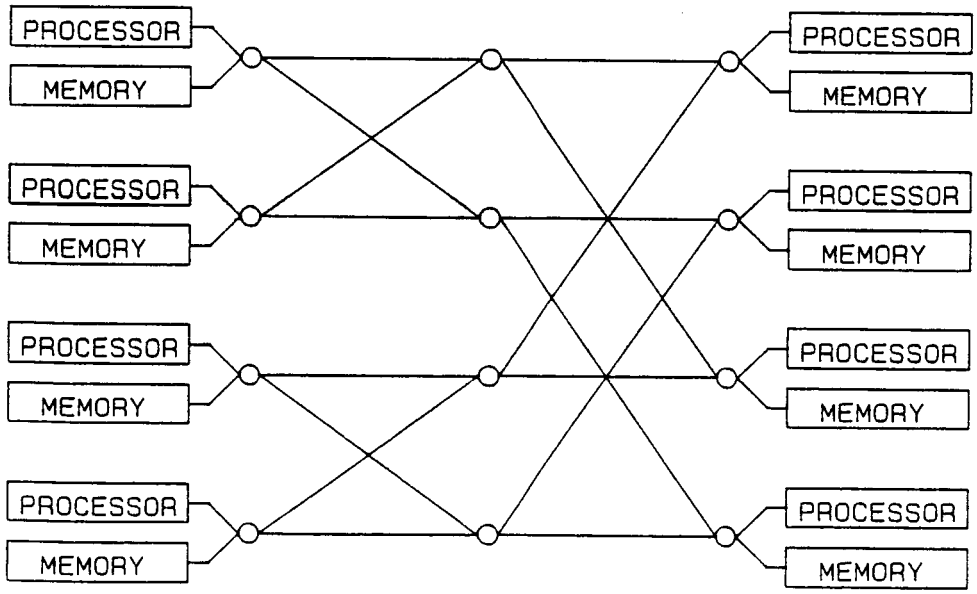
The two types of interconnection networks or topologies to be considered are the non-blocking crossbar switch and the fiber optic star. Crossbar switches offer the highest communication bandwidth and the most complex and costly design. Fiber optic stars offer higher communication bit rates than crossbar switches but only one PE may use the star network at a time. These interconnection networks are illustrated in figure 3.

If a system has a high degree of intercommunication between PEs, then a crossbar switch will offer the highest efficiency; on the other hand, if communication between PEs is low a fiber optic star will offer the best solution. As stated earlier, there is relatively little communication between tasks and what communication is present tends to be broadcast-type transfers to update state variables in the system; therefore, a fiber optic star would probably offer a more nearly optimal solution than the crossbar switch when all variables such as transmission format, cost, complexity, and transfer rates are considered.

Clustering is a technique in which PEs are grouped together with a dedicated interconnection network, and these groups or clusters of PEs are connected by a dedicated interconnection network. By creating levels in the interconnection network, clustering allows PEs in a cluster to operate on shared data with low communication overhead and provides hardware facilities for multiple groups of PEs to execute a



A FIBER OPTIC STAR



4 X 4 CROSSBAR SWITCH

Figure 3. Possible Interconnection Networks

tightly coupled process within their cluster without affecting the communications outside their cluster (Briggs and Hwang 1984). Figure 4 shows an interconnection network that uses fiber optic stars within a cluster and a fiber optic star connecting the clusters.

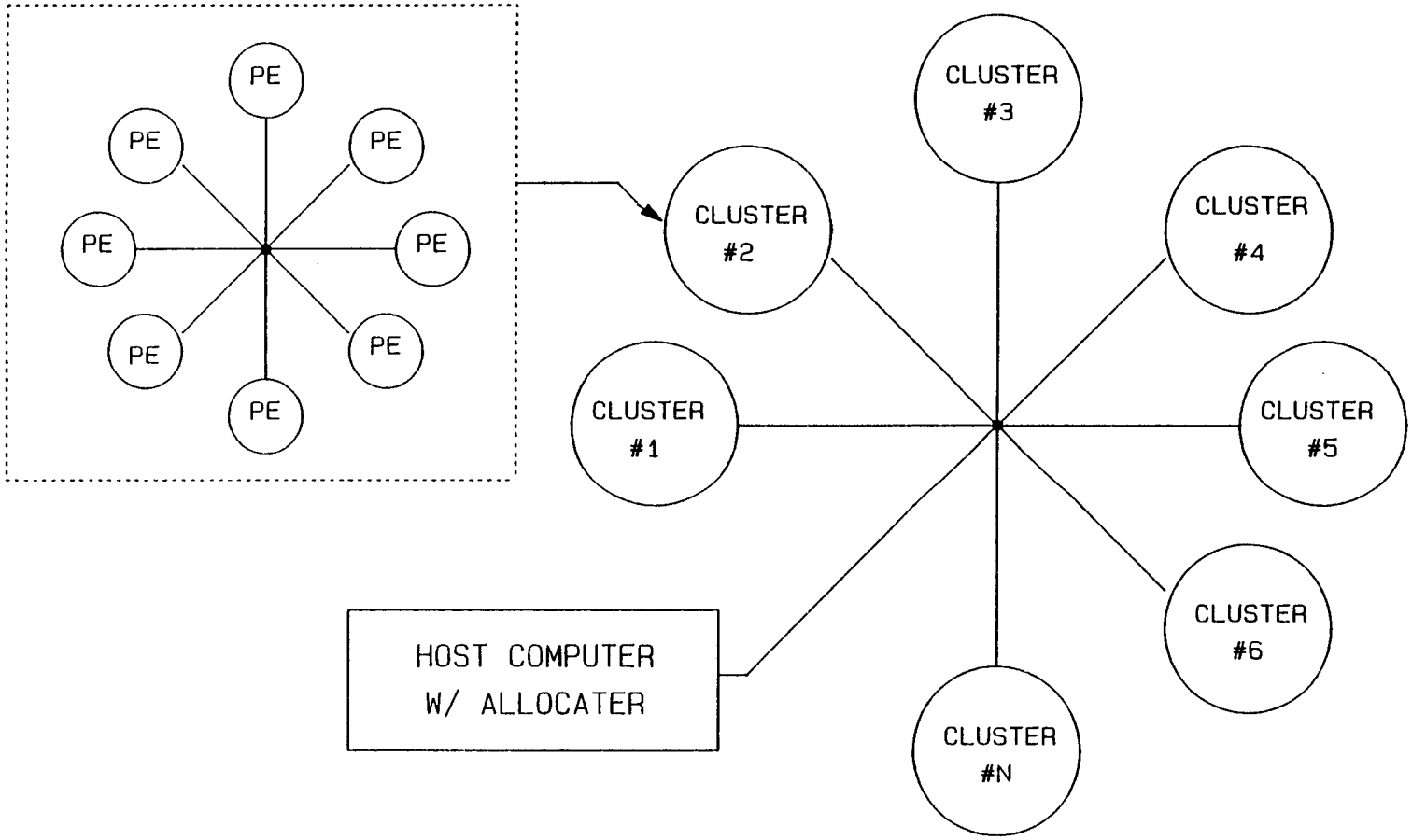


Figure 4. Clustered Network Using Fiber Optic Stars

CHAPTER 3

REAL-TIME INSTRUCTION EXECUTION WITH ACSL

In order to simulate complex systems in real-time, parallel processing will be incorporated into the architecture to boost execution rates to maximum levels. Parallelism will be implemented on two levels: the construct level and the program level. After examining the data flow graphs in appendix A, it does not take long to realize that very few of the ACSL constructs contain parallelism. Fortunately, one of the most important constructs in ACSL can be implemented with a parallel algorithm; that construct is the INTEG or integration instruction.

Parallelism on the program level is much more accommodating than parallelism on the construct level. Considering the fact that simulations executed on an analog computer are programmed in an inherently parallel manner, then it becomes clear that simulation programs written in ACSL can be mapped onto a parallel architecture in the same manner that simulations are mapped onto an analog computer architecture.

A direct-execution architecture offers several advantages over the traditional compiler approach to high-level language implementation with the largest advantage being in the form of more efficient code which results in faster program execution. In a direct-execution architecture, the compiler is eliminated altogether, and in its place an allocater is used to allocate segments of ACSL programs to the various PEs in the system. Resident at each PE are the hand-written assembly language routines to execute all ACSL constructs which will

result in the most efficient programming possible. Although it is beyond the scope of this paper to design the allocater, an attempt will be made to specify its requirements and describe its basic operation.

Parallelism on the Construct Level

ACSL constructs have been classified into one of three different categories. These categories are constructs with no inherent parallelism, constructs approximated with a finite term series (such as trigonometric functions), and constructs with inherent parallelism. Table 1 shows all constructs in their appropriate category. Their data flow graphs and microcoded routines can be found respectively in appendix A and appendix B.

Constructs in category I offer no parallelism and are executable on one PE; in fact, several of these construct routines may be allocated to one PE without overfilling that PE's program memory. Constructs in category I range from simple boundary checks to simple calculations.

Constructs in category II again offer no parallelism in a course-grained system but can be computed very efficiently by the use of a floating point processor that is optimized for factored polynomial evaluation. This point is discussed further in chapter 4.

Constructs in category III have useful amounts of inherent parallelism which are exploitable in a course-grained system. The most important construct in this category is the integrate instruction. In order to take advantage of parallelism, the integrate construct will use a second order parallel predictor-corrector algorithm. This algorithm is simply a restructuring of the traditional predictor-corrector method to allow predicting of the n+1 value while at the same time correcting

TABLE 1

CATEGORIZED ACSL CONSTRUCTS

CATEGORY I

ACSL INSTRUCTIONS WITH NO PARALLELISM PRESENT IN A COURSE-GRAINED SYSTEM.

ABS - ABSOLUTE VALUE.
AMOD - REMAINDER OF MODULUS.
BCKLSH - BACKLASH OR HYSTERICES.
BOUND - LIMIT A FUNCTION.
DBLINT - LIMIT DISPLACEMENT TERM OF FUNCTION.
DEAD - CREATE DEADSPACE.
DELAY - DELAY WITH RESPECT TO TIME.
DERIVT - 1ST ORDER DERIVATIVE.
DIM - POSITIVE DIFFERENCE.
FCNSW - FUNCTIONAL SWITCH.
GAUSS - CREATE NORMALLY DISTRIBUTED RANDOM VARIABLE.
HARM - CREATE A SINUSOIDAL FUNCTION.
IABS - ABSOLUTE VALUE OF AN INTEGER.
IDIM - POSITIVE DIFFERENCE OF INTEGERS.
INT - INTEGERIZE F.P. VALUE.
ISIGN - APPEND A SIGN.
LIMINT - LIMIT INTEGRATION.
LSW,RSW - LOGICAL AND REAL SWITCH FUNCTIONS.
MOD - REMAINDER OF AN INTEGER DIVISION.
PTR - POLAR TO RECTANGULAR CONVERSION.
PULSE - GENERATE A PULSE TRAIN.
QNTZR - QUANTIZE A VARIABLE.
RAMP - LINEAR RAMP FUNCTION GENERATOR.
RTP - RECTANGULAR TO POLAR CONVERSION.
SIGN - APPEND A SIGN.
STEP - GENERATE A STEP FUNCTION.
UNIF - UNIFORM RANDOM NUMBER SEQUENCE.
ZHOLD - ZERO ORDER HOLD.

CATEGORY II

ACSL INSTRUCTIONS APPROXIMATED WITH A FINITE TERM SERIES.

ACOS - ARC COSINE.
ALOG - NATURAL LOGARITHM.
ASIN - ARC SINE.
ATAN - ARC TANGENT.
COS - COSINE.
EXP - NATURAL EXPONENT.
EXPF - SWITCHABLE EXPONENTIAL.
SIN - SINE.
SQRT - SQUARE ROOT.
TAN - TANGENT.

TABLE 1--Continued

CATEGORY III

ACSL INSTRUCTIONS WITH PARALLELISM:

AMAX0, AMAX1, MAX0, MAX1 - INTEGER AND FLOATING POINT MAXIMUM VALUE ROUTINES.

AMIN0, AMIN1, MIN0, MIN1 - INTEGER AND FLOATING POINT MINIMUM VALUE ROUTINES.

INTEG - INTEGRATION.

for the n value (Liniger, Werner, and Miranker 1966). Using this method, a speed increase factor close to two can be realized.

In addition to a parallel predictor-corrector method, a fourth order Runge-Kutta integration method will also be programmed (Ralston and Wilf 1965). Although basically a sequential process, the coefficients K1, K2, K3, and K4 of the Runge-Kutta algorithm can be computed for sets of simultaneous equations concurrently, thereby making the execution time for a system of N equations on a parallel processing computer approximately equal to the execution time for a system with a single equation on a sequential machine.

Parallelism on the Program Level

As stated at the beginning of this chapter, ACSL programs can be mapped directly onto a parallel architecture since simulations are typically executed on an analog computer, and analog computers tend to incorporate a large amount of parallelism. This is best demonstrated with an example using the Armstrong Cork Benchmark (Hannaver 1986).

An ACSL program called the Armstrong Cork Benchmark is shown in table 2. A restructured data flow graph of this program is shown in

TABLE 2

ARMSTRONG CORK BENCHMARK

```
INITIAL

     CONSTANT TEND = 50.0E-6

     CONSTANT K1 = 1.0E-14, K2 = 1.0E6, K3 = 1.0E3, K4 = 1.0E6
     CONSTANT K5 = 1.0E-2, K6 = 1.0E-5, K7 = 1.0E5, K8 = 1.0E6
     CONSTANT K9 = 1.0E-3

     CONSTANT X10 = 24., X20 = 0., X30 = 0., X40 = 0.
     CONSTANT X50 = 0., X60 = 0.

     MINTERVAL MINT = 1.0E-7
     MAXTERVAL MAXT = 1.0

END

DYNAMIC

     DERIVATIVE
          X1DOT = -K1*X1 - K3*X1*X4 - K7*X1*X3
          X2DOT = -K2*X2 + K1*X1 + K3*X1*X4 + K7*X1*X3 + K9*X4
          X3DOT = K6*X5*X5 - K7*X1*X3 - K8*X3*X4
          X4DOT = K2*X2 - K3*X1*X4 - K4*X4*X4*X4 + K6*X5*X5 -
                  K8*X3*X4 - K9*X4
          X5DOT = K3*X1*X4 - K5*X5*X5 - K6*X5*X5
          X6DOT = K4*X4*X4*X4 + K5*X5*X5 + K8*X3*X4

          X1 = INTEG(X1DOT, X10)
          X2 = INTEG(X2DOT, X20)
          X3 = INTEG(X3DOT, X30)
          X4 = INTEG(X4DOT, X40)
          X5 = INTEG(X5DOT, X50)
          X6 = INTEG(X6DOT, X60)

          EPS = (X1 - X10) + X6 + (X2 + X4 + X5)

          TERMT(T.GT.TEND)
     END
END

TERMINAL
END
```

figure 5. The graph in figure 5 can be thought of as a set of 6 integrators operating in parallel. One possible allocation algorithm might be to allocate the algebraic equation X1DOT and the integrator X1

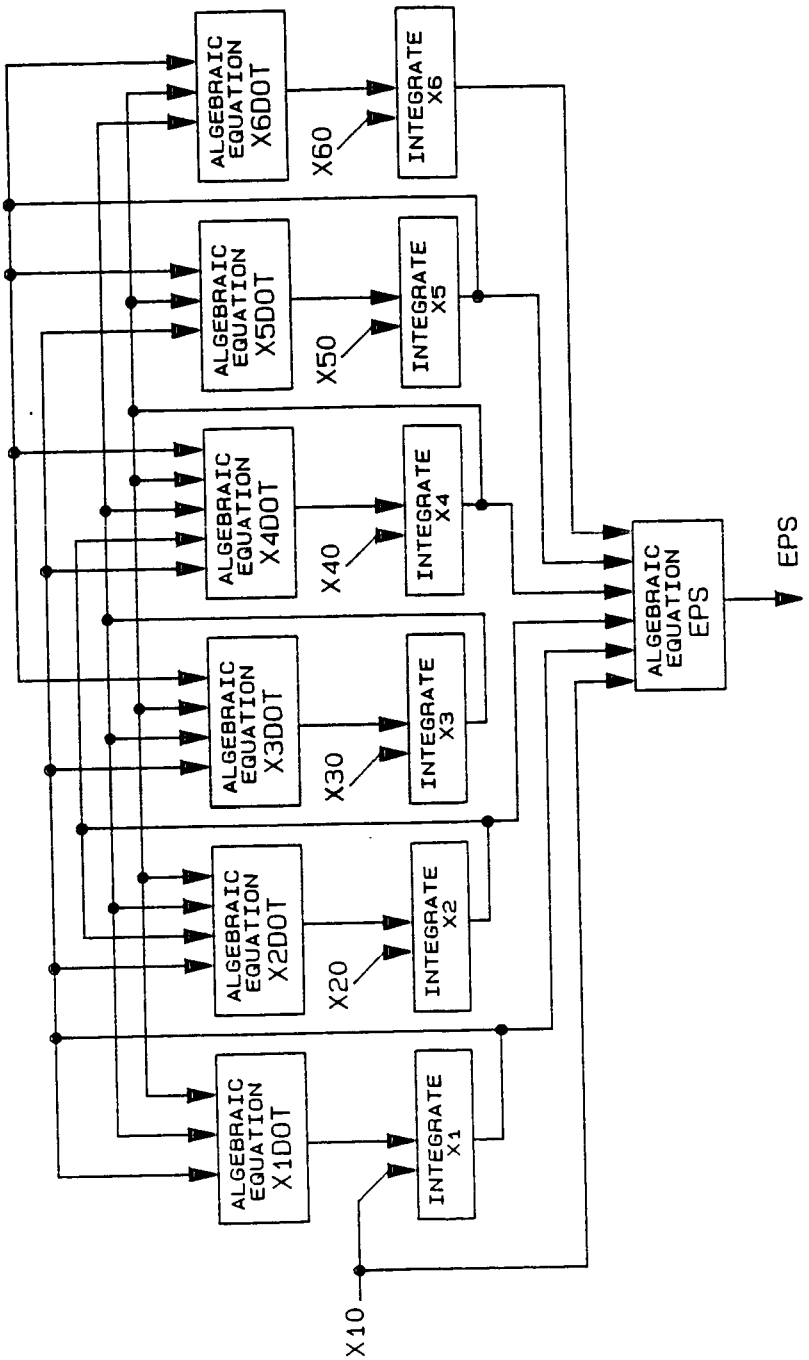


Figure 5. Data Flow Graph of Armstrong Cork Program

to a cluster of PEs, the equation X2DOT and the integrator X2 to another cluster of PEs, and so on until all six equations and integrators are allocated. These six clusters of processors would then compute their equations, perform their integrations, and update their results at the appropriate communication interval all in parallel. An additional PE would be required to accumulate the program output EPS. With this configuration and the use of parallel integration methods, an execution speed increase of up to twelve could be realized over the usual uniprocessor approach.

Allocater Requirements

For real-time operation, the allocater has to allocate ACSL constructs in the most efficient manner, reduce and factor equations for maximum execution speed, and optimally schedule the interprocessor communications.

Resource and Construct Allocation

System resources must be allocated in an optimum manner to obtain maximum system throughput. Instead of using main memory locations to hold variables, the allocater will assign internal CPU and FPU registers to frequently used variables. Doing so will allow maximum execution speeds since most variables will be immediately available to CPU.

To maximize execution speed, ACSL constructs must be allocated on the cluster level and the PE level in the most efficient manner. In general, independent functions should be allocated in a way to allow them to be executed in parallel; in addition, tasks with a moderate to high degree of processor intercommunications should be allocated to a cluster of PEs, thus allowing the required communications between

processors to proceed without hindering other processor's intercommunications. An example of this would be to allocate an integration and its associated equation to a cluster of PEs as described in the section Parallelism on the Program Level. Two or more PEs in the cluster could be used on a parallel integration algorithm and one or more of the PEs could be used to evaluate the derivative function. The optimum number of PEs in a cluster would depend on the complexity of the program.

Expression Reduction and Factoring

The allocater will be responsible for reducing algebraic expressions down to the form that will allow maximum execution speed. This process may include factoring a polynomial to allow rapid multiply/accumulate sequences. The constructs involving finite term series approximations are programmed in this manner. To illustrate factoring, examine the following equation for the exponential function (Beyer 1984):

```
EXP[X] = 1 +X +A0*X**2 +A1*X**3 +A2*X**4 +A3*X**5 +A4*X**6 +A5*X**7.
```

It may be factored to allow computation without generating higher powers of X in the following manner:

```
EXP[X] = 1+X[1+X[A0+X[A1+X[A2+X[A3+X[A4+X[A5]]]]]]].
```

This form allows rapid computation of an exponential with floating point units that incorporate a multiply/accumulate ALU.

Interprocessor Communication Scheduling

The allocater will be responsible for determining what time periods the PEs will have data ready to transmit. Using this information, the

allocater will schedule the order in which intracluster PEs will transmit to other intracluster PEs and the order in which clusters will transmit data to other clusters. The allocater is able to schedule data transfers, because the execution times of all the construct routines are known. Once the order of all data transfers is known, the allocater loads information into every I/O processor telling it which data words to use, what order they arrive, and where the data words go in the PEs memory. For example, the allocater might tell a particular I/O processor that the fourth word seen on the star after the start of a calculation interval is state variable X1 which it needs for its calculations. The I/O processor would read X1 (and any other variables it required) and ignore all others. This process would repeat every calculation period.

The reason this complex scheduling scheme was devised is to minimize the communication overhead associated with transferring a word of data between PEs. If data packets were used that contained addressing or destination information, it would significantly increase the communication delays in the system. Using the scheduling technique allows the use of data packets that contain little or no overhead characters, only 32 bits of data.

Real-Time Data Transfer

In order for a PE to transmit a word of data to another PE, the only action taken by the transmitting PE is to write the data word to the I/O processor. The I/O processor knows which data word it just received from the predetermined order assignments. The I/O processor then formats the data into a packet, waits for that data word's scheduled turn on the network, and then places it on the fiber optic

star network. If the data is scheduled to be received by an intra-cluster PE, that PE's I/O processor reads the data word and deposits it into the proper location in that PE's memory. If the transfer is inter-cluster, it will be read by the cluster I/O processor and placed on the intercluster fiber optic star for reception by the proper cluster at the scheduled time.

When a data word is received by the I/O processor for use by the PE, the I/O processor writes the data word to a predetermined memory location and signals the PE by activating an interrupt line. The interrupt causes the PE to retrieve the data word atomically with the LOADSET instruction. The LOADSET instruction reads a word of memory and writes back all ones to the same location after the read. With this technique, the I/O processor can determine if the PE has read the data word before updating the memory location with a new value. When all the operands have been received and loaded by the interrupt routine, the interrupt routine signals the executing task by setting a condition code to the boolean true value. The executing task simply checks this condition code, and when true, continues with the program execution.

Program Execution with Direct-Execution Architectures

Direct-execution architectures offer several advantages over other architectures that use translators and compilers in that the direct-execution architecture has no compilation overhead, offers single-copy program storage, and has a high degree of interactiveness (Milutinovic 1988). A disadvantage of a direct-execution architecture is its lack of flexibility. Most direct-executions languages are implemented entirely in hardware thus representing a nonoptimal hardware/software tradeoff, except in specific applications. For example, it would not be advan-

tageous to use a direct-execution architecture developed for Ada in a system used to execute FORTRAN code. Although the architecture discussed in this thesis was designed to execute ACSL, it is flexible enough to implement other host languages by simply re-writing the micro-coded routines to execute whatever high-level language desired.

CHAPTER 4

PROCESSING ELEMENT CONFIGURATION

A processing element (PE) will be composed of a CPU, a FPU, high-speed instruction memory, main memory, and an I/O processor. The CPU will be a state-of-the-art microprocessor capable of sustained high MIPS rates. The FPU (floating point unit) will be required due to the numerically intensive tasks associated with ACSL. The high-speed instruction memory will hold the microcoded ACSL construct routines the CPU will execute, allowing the CPU to execute at full speed without wait states. Main memory will be made up of dynamic RAM and EPROM which will hold the microcoded routines for all ACSL constructs. The I/O processor is responsible for monitoring the interconnection bus and relieving the CPU from the overhead associated with interprocessor communications. A block diagram of a PE is shown in figure 6.

Execution Flow In The Processing Element

Before an intelligent choice can be made as to what type processing element to use, there must be an understanding as to the execution environment the PE will be operating in. Figure 7 shows a typical direct-execution architecture known as the University of Maryland approach. Code to be executed is stored in the program memory, and data variables are stored in the data memory. At execution, the lexical processor scans the program memory to determine what high-level language tokens it is to execute. The lexical processor then places the tokens

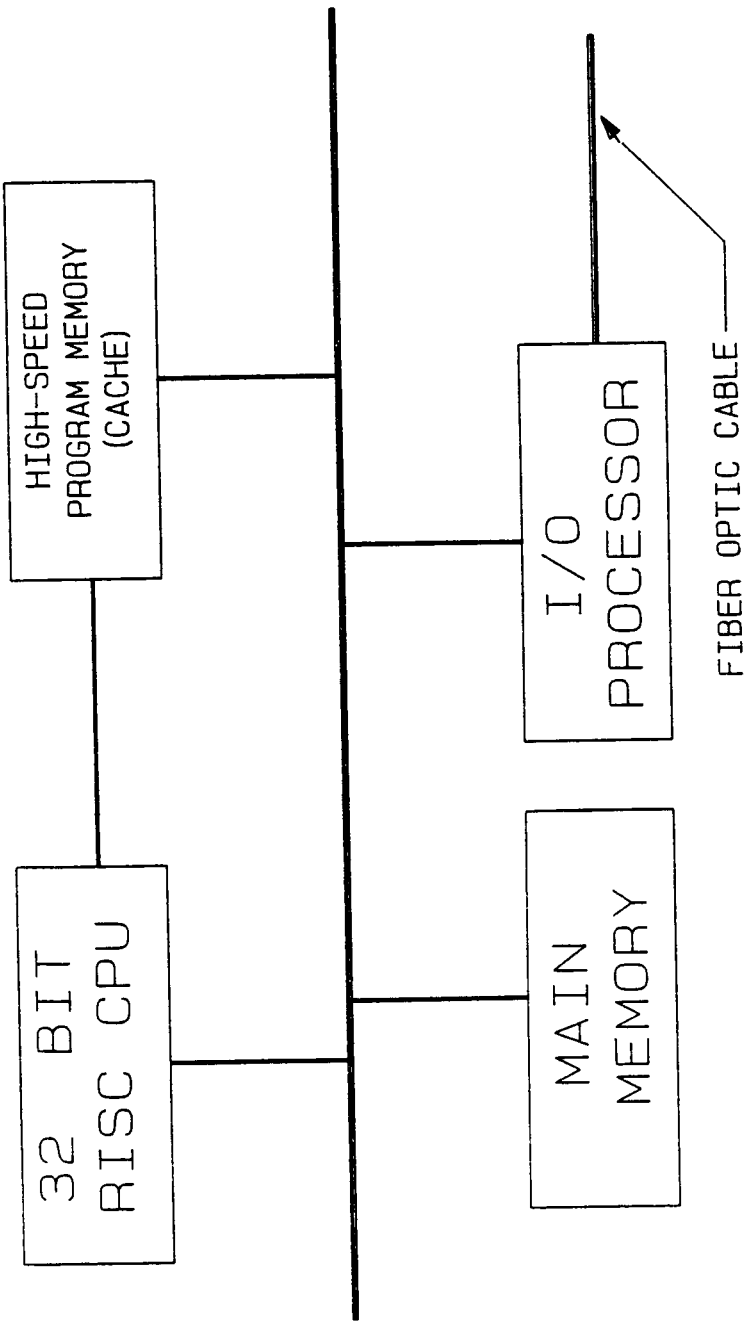


Figure 6. Processing Element

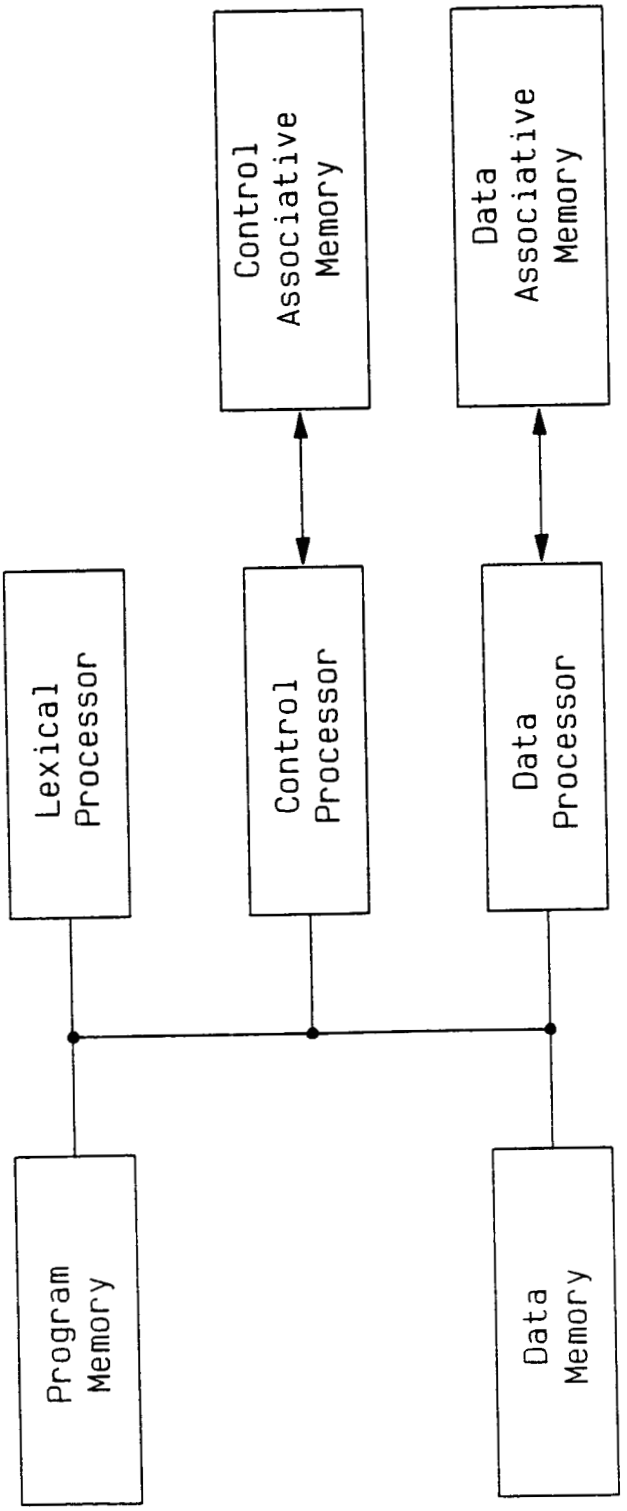


Figure 7. The University of Maryland Direct-Execution Architecture

into a token register for execution by the control processor or data processor. The data processor will execute tokens corresponding to data manipulations such as multiplication and shifting, and the control processor executes tokens that change program flow such as GOTO and IF THEN instructions (Milutinovic 1988).

The architecture designed to implement ACSL will be similar to the University of Maryland approach, but will deviate from it in several ways in order to simplify the design and increase program execution. The ACSL direct-execution architecture will use only one processor to execute program control instructions as well as data manipulation instructions, although a floating point processor will be included to assist in the calculation of floating point operands. To obtain the maximum execution speeds, all lexical analysis will be performed by the allocater prior to the start of execution. The tokens to be executed will be resident in the CPU when execution starts. This is possible due to the fact that ACSL programs execute the same code repetitively as shown by the bold line (the primary loop) in figure 8. The code in the DERIVATIVE section is simply executed over and over, incrementing the time variable each pass until the terminate conditions are met. Performing lexical analysis before execution starts eliminates the need for a lexical processor in hardware, thus simplifying the design and increasing throughput by eliminating any delays associated with having the CPU wait for tokens.

When a PE receives (in the preprocessing stage) an ACSL construct it is to execute or an operand it will use in execution, that PE transfers the microcoded routine from main memory into high-speed static RAM or deposits the operand into an internal CPU register. When program

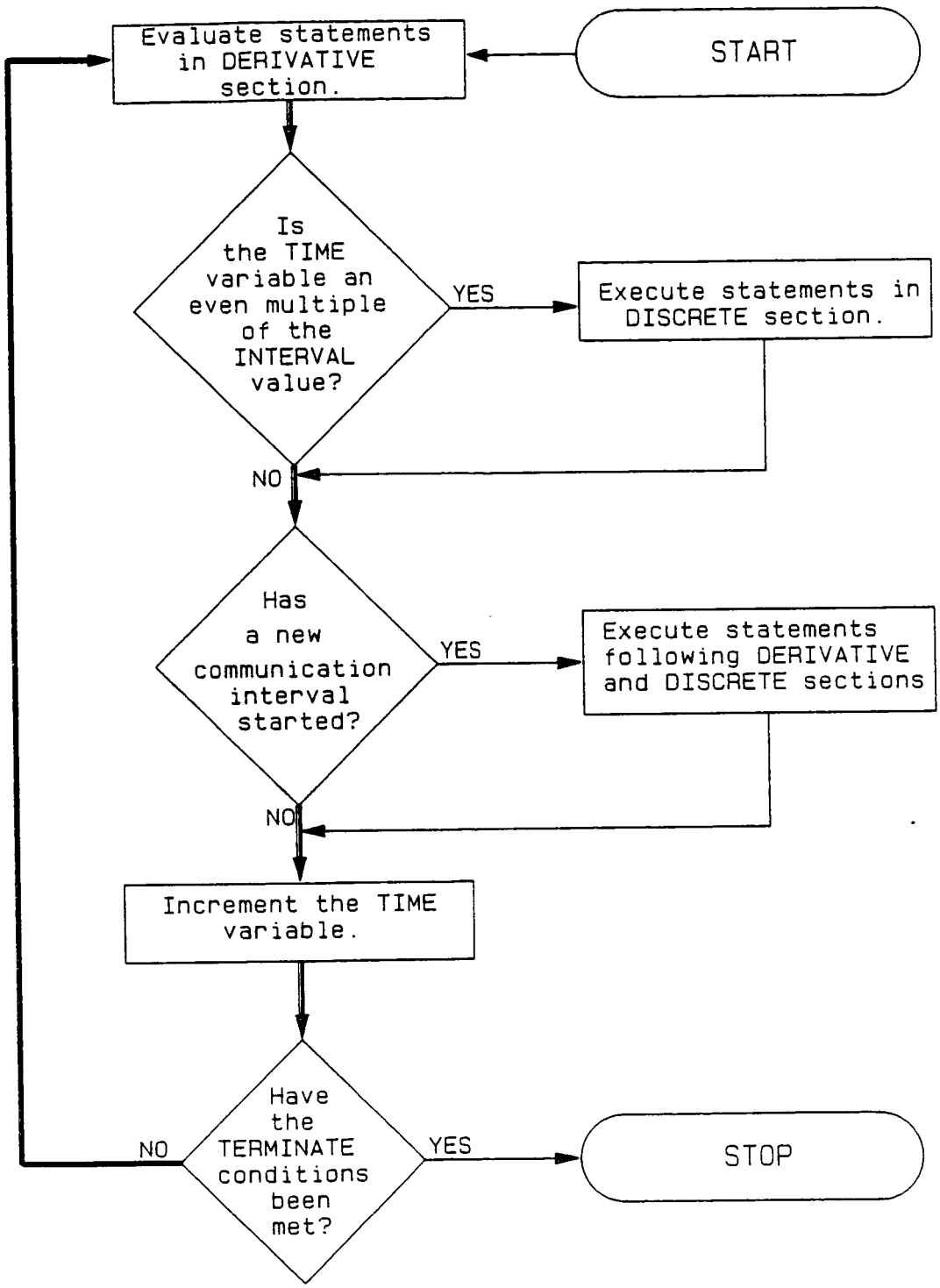


Figure 8. Instruction Execution Flow

execution begins, the PE will be executing instructions entirely from high-speed RAM with all operands contained in internal CPU registers, thus allowing the fastest possible execution by eliminating access to slow main memory.

CPU

Microprocessors examined will be limited to the new families of 32 bit machines in order to obtain the most performance per processing element (PE). Of the available 32 bit microprocessors there are two basic types, complex instruction set (CISC) or reduced instruction set (RISC) microprocessors. Our search will concentrate on RISC type processors since they are characterized as having a large register set, being able to execute one instruction per clock cycle and having a high MIPS rate. RISC processors usually use a hard-wired instruction decoder that allows it to execute one instruction per clock cycle; on the other hand, CISC processors usually pay a performance penalty for supplying more complex instructions in the form of multiple clock cycles per instruction. In most cases it simply takes longer to decode and execute complex variable length instructions, and this inefficiency causes even simple operations in a CISC processor to take multiple clock cycles to execute--operations that a RISC processor would execute in one clock cycle (Toy, Wing, and Zee 1986).

Microprocessor Survey

All the microprocessors listed in table 3 have one feature in common, except the 86C010 ARM, and that is they all use a dual bus Harvard architecture. Harvard architectures significantly improve performance by allowing data and instruction accesses simultaneously.

TABLE 3

COMPARING 32 BIT RISC MICROPROCESSORS

NAME	MANUF.	CLOCK SPEED	CPU REGISTERS	MIPS
29000	AMD	25 MHz	192	25.0
TRANSPUTER	INMOS	20 MHz	3	10 - 20
CLIPPER	FAIRCHILD	33 MHz	32	5 - 33
86C010 ARM	VLSI TECH.	8 MHz	25	4.0
29300	AMD	11.11 MHz	65	11.11
74AS88XX	TI	20 MHz	65	20.0

AMD 29000

The AMD 29000 has built-in support for both a data and an instruction cache with the cache memory located externally from the processor. It is said to be targeted toward general purpose CPU applications such as workstations and personal computers. Operating at 25 MHz, Advanced Micro Devices claims the 29000 offers about 12 times the performance of a VAX-11/780 for integer and systems code. The generous 192 register file acts as a data cache for a moderate number of operands while giving the ability to read-modify-write data in a single clock cycle. The AMD 29000 has a 32 bit fixed width instruction set. Large, fixed length instructions encode programs less efficiently (in terms of memory used) than small, variable-length instructions, but they can be processed much faster. The AMD29027 FPU is available to increase floating point computations greatly over software routines (Johnson 1987b).

Inmos Transputer

The Inmos Transputer was designed with parallel processing in mind.

It is a high-integration machine which has four high-speed (20M bps) serial channels for communication with other processors, two timers, an 8 channel DMA controller, and a dynamic ram controller on chip. Systems have been built using over 100 Transputers that showed a linear increase in computing speed for each Transputer added. A disadvantage of the Transputer architecture for this application is the fact that the I/O processor is located internally in the CPU and must be explicitly programmed. For maximum speed, an external I/O processor should be used to relieve the CPU of the overhead associated with formatting the message/data, computing where the data is to be send, etc. The Transputer must take time to program the serial I/O channels with such information--time that could have been spent by the CPU in executing an ACSL program. A new upgraded Transputer with a built-in floating point processor has been announced by Inmos and should be available soon. This will make the Transputer family more attractive to number-crunching applications, but for now it will not be considered further in this application (Cushman 1987).

Fairchild Clipper

The Fairchild Clipper represents to some the state of the art in 32 bit microprocessors. This three chip set is actually a blend between CISC and RISC in that it contains a microprogram ROM to execute complex instructions, but this ROM is by-passed when the Clipper executes a more primitive instruction. This offers the high MIPS rate of RISC processors while still enjoying the sophisticated instructions of a CISC. Another feature of the Clipper is that it contains a complete 4K byte data cache, a 4K byte instruction cache, and a floating point processor on-board. The Clipper's register file contains 32 general

purpose registers that can be used for data or addressing, and eight 64 bit registers that are dedicated to floating point calculations. The Clipper instruction set contains 101 hardwired single clock cycle instructions and 67 microprogrammed complex instructions. Instruction lengths vary from 16 to 64 bits wide in multiples of 16 bits (Hunter 1987).

VLSI 86C010

The VLSI 86C010 is intended to be a low cost RISC 32 bit microprocessor. Due to its standard von Neumann architecture, slow clock speed, and small register file, it will be a low performance device as well and will not be considered for this architecture (Cushman 1987).

TI 74AS88XX and AMD 29300

The remaining two processor families, the TI 74AS88XX and the AMD 29300, are very similar architectures, and many designers mix and match various family members from both manufacturers when designing a system. Both of these processors are fixed-width 32 bit bit-slice microprocessors. A bit-slice CPU is composed of various parts that let a user configure his architecture for his application. Using a bit-slice CPU does increase your parts count, but this is offset by the fact that a bit slice system usually offers superior performance to that of a fixed architecture microprocessor. As in a classic RISC machine, a bit slice processor executes one microinstruction per clock cycle. The designer has the option of including complex instructions in the architecture by writing a microprogram (with the primitive microinstructions) to implement the complex function (Cushman 1987).

Microprocessor Selection

The three most impressive microprocessors in the survey are the AMD 29000, the Fairchild Clipper, and the TI 74AS88XX family. All three offer similar instruction sets. The Clipper does offer more complex instructions than the others, but this is offset by the fact that these instructions take multiple clock cycles to execute. As far as register files are concerned, all three processors offer a large number of registers with the Clipper having 32, the 74AS88XX having an expandable 65 word register file, and the AMD 29000 boasting an impressive 192 general purpose 32 bit registers. The last factor to consider before selecting a microprocessor is the execution speed or MIPS rate of each processor. Given that all three processors can execute a simple instruction in one clock cycle, the limiting factor in performance becomes the access time to the instruction cache or instruction memory. A processor cannot execute instructions faster than it fetches them.

The Clipper contains a 4K byte instruction cache configured as a four-way set associative cache with a quadword line buffer. If the total access time for the quadword line buffer is 60nS and the total access time for the main cache memory is 120nS, then depending on the instruction length (16 to 64 bits), the average access time for in-line code would be given by table 4 (Hunter 1987).

Examining these calculations shows that even if all the Clipper instructions were 16 bits long, the highest MIPS rate possible would be 15 MIPS. This figure does not take into account the time necessary to load the quadword line buffer or the main cache memory. Unless the cache mechanism can be disabled and the Clipper allowed to fetch one instruction word per clock cycle, it appears that in this application

TABLE 4

AVERAGE INSTRUCTION ACCESS TIMES FOR CLIPPER

INSTRUCTION SIZE	ACCESSES TO LINE BUFFER @60nS EACH	ACCESSES TO MAIN CACHE @120nS EACH	TOTAL AVG. ACCESS TIME
16 BIT	8	1	66.67nS
32 BIT	4	1	72nS
64 BIT	2	1	80nS

(where the program will be executed totally out of high-speed static RAM) the cache system will deteriorate performance rather than enhance it.

The AMD 29000 offers several access protocols: simple access, pipelined access, and burst-mode access. All of the access protocols will fetch instructions at a 25MHz rate, but the burst-mode is easier to implement, since there are not as many address transfers or decodes to perform (Johnson 1987b).

The TI 74AS88XX uses a simple pipelined approach to access program memory. A microsequencer places an address on the microprogram memory, and the same clock edge that updates the microsequencer address output latches the previously addressed output from the microprogram memory into the instruction register. This allows instructions to be executed and fetched at a maximum rate of 20MHz (Texas Instruments, Inc. 1985).

After examining the results of the above analysis and the features contained in each of the microprocessors, it appears that the AMD 29000, with its 192 general purpose registers and 25 MIPS execution rate, offers the best solution. A block diagram of a PE constructed with an AMD 29000 is shown in figure 9 (Johnson 1987a). The TI bit-slice

processor family offers approximately the same performance (or possibly slightly higher, depending on the efficiency of its FPU) as the AMD 29000, but due to the complexity and size of this bit-slice CPU, it was not selected. The Clipper performs at its best when used in systems with several layers of memory hierarchy, ranging from very fast to extremely slow. The Clipper's elaborate cache and memory management unit decreases the effective access time from 500nS for dynamic RAM to around 100nS, a 500% improvement in performance, but, the Clipper does not seem well suited to this particular application (Hunter 1987).

Understanding The AMD 29000

For those not familiar with the AMD 29000, there are a few details about its programming that should be understood. Due to its pipelined architecture, the AMD 29000 uses a delayed branch mechanism. With a delayed branch, the instruction immediately following a branch instruction will always be executed. In the case where a useful instruction can be placed after a branch instruction, the branch instruction has an execution time of one clock cycle; otherwise, a NOP instruction will have to follow the branch, giving the branch instruction an effective execution time of two clock cycles.

The AMD 29000 has a unique way of handling conditional instructions. Instead of having a flag register which all conditional instructions use, the AMD 29000 allows conditions for instructions to come from any general purpose register. Certain instructions set true or false boolean values in a general purpose register, values that conditional branch instructions can use at a later time to determine whether to take a branch or to continue execution (Johnson 1987b).

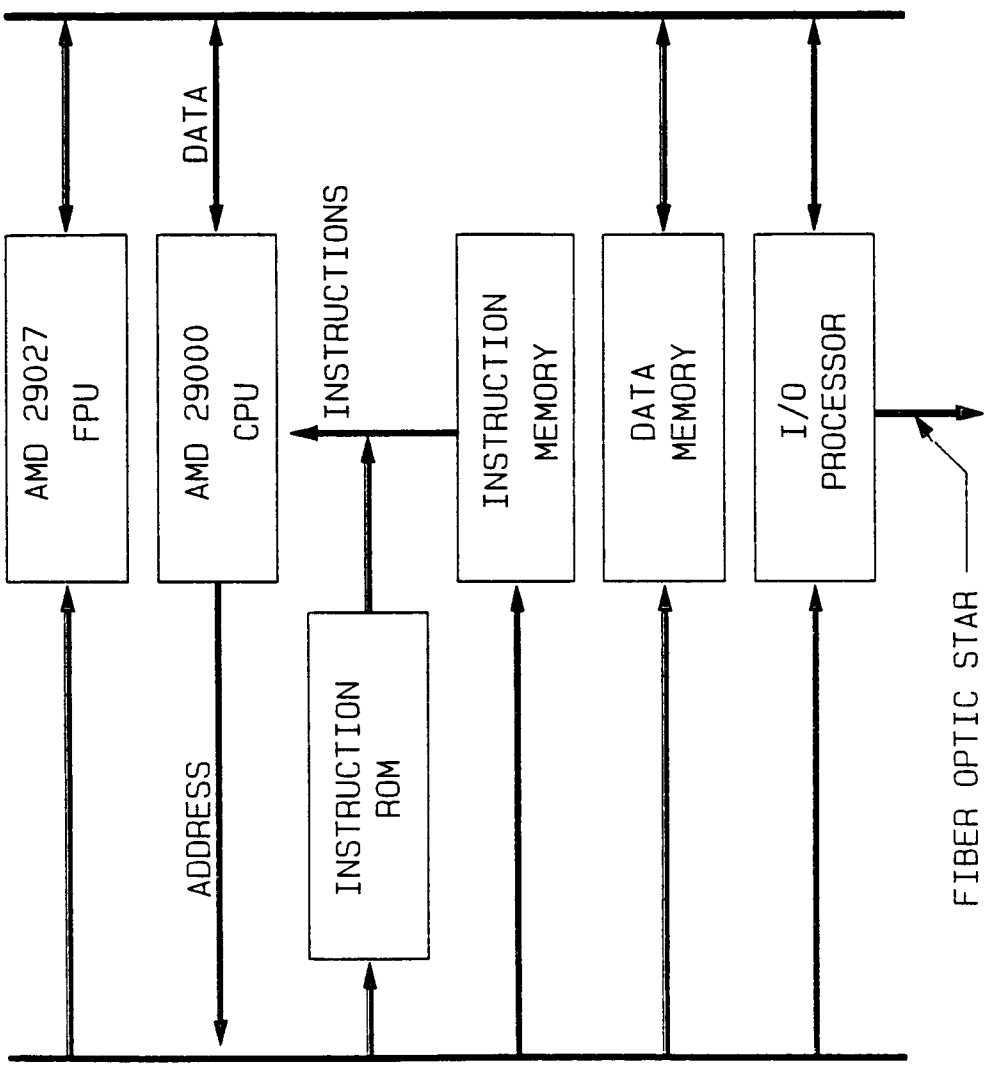


Figure 9. AMD 29000 Based Processing Element

Microprogram Timing Analysis

The following paragraphs will examine the execution times of the AMD 29000 assembly language ACSL construct routines, shown in Appendix B, and specify the assumptions and conditions used to obtain these speeds. A summary of the execution times and the average MIPS rates for the ACSL construct routines is shown in table 5.

TABLE 5

MICROPROGRAM TIMING RESULTS

CONSTRUCT	INSTRUCTIONS EXECUTED	AVERAGE MIPS	EXECUTION TIME (nS)
ABS	1	25	40
ACOS	35	12.5	2800
AINT	7	12.5	560
ALOG	62	12.8	4840
AMAX0	7*CNT+12	varies	(7*CNT+21)*40
AMAX1	7*CNT+12	varies	(10*CNT+21)*40
AMIN0	7*CNT+12	varies	(7*CNT+21)*40
AMIN1	7*CNT+12	varies	(10*CNT+21)*40
AMOD	40	12.99	3080
ASIN	31	12.5	2480
ATAN	77	12.5	6160
BCKLSH	24	15.38	1560
BOUND	20	16.67	1200
COS	28	12.5	2240
DBLINT	18	16.08	1120
DEAD	21	14.58	1440
DELAY	12	21.42	560
DERIVT	41	13.31	3080
DIM	10	15.63	640
EXP	31	12.5	2480
EXPF	39	13.0	3000
FCNSW	15	21.43	700
GAUSS	153	15.61	9800
HARM	47	13.06	3600
IABS	5	12.5	400
IDIM	4	25	160
INT	5	12.5	400
INTEG:			
RUNGE-KUTTA	43+4*FUNCTION	12.5	3280+4*FUNCTION
PREDICTOR	14+FUNCTION	12.5	960+FUNCTION
CORRECTOR	16+FUNCTION	varies	1160+FUNCTION+DELAY
ISIGN	5	12.5	400
LIMINT	16	15.39	1040

TABLE 5--Continued.

CONSTRUCT	INSTRUCTIONS EXECUTED	AVERAGE MIPS	EXECUTION TIME (nS)
LSW	3	25	120
MAX0	7*CNT+9	varies	(7*CNT+15)*40
MAX1	7*CNT+16	varies	(10*CNT+29)*40
MIN0	7*CNT+9	varies	(7*CNT+15)*40
MIN1	7*CNT+16	varies	(10*CNT+29)*40
MOD	55	25	2200
PTR	70	12.5	5600
PULSE	31	16.15	1920
QNTZR	60	17.05	3520
RAMP	10	15.63	640
RSW	3	25	120
RTP	353	13.81	25560
SIGN	5	12.5	400
SIN	32	12.5	2560
SQRT	245	15.09	16240
STEP	10	16.67	600
TAN	32	12.5	2560
UNIF	17	14.66	1160
ZHOLD	3	25	120

Assumptions Used In Analysis

The following assumptions were made in the analysis of the ACSL construct routines:

1. Operands (except where noted) are contained in internal registers in the AMD 29000.

2. There will be a 100% hit ratio in the Branch Target Cache.

3. Instruction memory will accommodate one cycle accesses.

4. Load and store instructions require two clock cycles for execution, and all other instructions used require only one clock cycle.

5. All floating point operands are 32 bit single precision values.

Since there are 192 general purpose registers in the AMD 29000 and for a large number of PEs there will be a relatively small number of operands per PE, the assumption that all operands will be held in

internal CPU registers is not unreasonable. This assumption is verified by examining the microcoded routines in appendix B. There is not a single routine that uses more than a ten registers.

The AMD 29000 contains a Branch Target Cache which holds information regarding the 32 most recent branches. The first time a branch is executed the four instructions at the target location are saved in the Branch Target Cache. When the branch is executed again, the processor pipeline is filled with instructions from the Branch Target Cache, allowing the processor to proceed without having to wait for the pipeline to refill. Since ACSL programs execute the same code in the derivative section repetitively and the construct routines tend to contain in-line code, it is safe to say that branches taken will be contained in the Branch Target Cache.

In order to have memory fast enough to allow one-cycle accesses, static RAM memory with access speeds of 20nS or less will have to be used. This does not present a problem since there are several memory devices on the market today with speeds of 15nS to 20nS.

All AMD 29000 instructions (except LOADM and STOREM) are designed to execute in one clock cycle. Unfortunately, the LOAD and STORE instructions will require two clock cycles to execute when instructions are being fetched at a rate of 25 MHz (one per clock cycle), because the address bus is shared between data operations and instruction fetches. The processor will require one clock cycle to generate the LOAD or STORE instruction address and another clock cycle to generate the operand address. In applications where multiple clock cycles are required to access memory, the loading and storing of data operands will be transparent, since the operand address generation will be performed during

the the time periods (referred to as wait states) the CPU is waiting for instruction memory.

Using 32 bit floating point operands will provide sufficient resolution and accuracy for the majority of applications. If double precision operands are required, execution times will not significantly increase, since the AMD 29027 FPU performs double precision operations in the same amount of time as single precision operations. The only additional requirements will be the time necessary to load and unload the extra words to and from the FPU and the extra storage required for the larger operands.

An Optimal CPU/FPU

After examining the characteristics of the AMD 29000 microprocessor, it becomes obvious that a theoretical processor with different features could significantly improve the performance of this architecture. Possible improvements to the processor include the following:

1. Separate address buses for instructions and data.
2. Integral floating point unit.
3. Non-pipelined operation, both in the CPU and the FPU.

The only factor keeping all construct routines from operating at 25 MIPS is the fact that two clock cycles are required for a LOAD and a STORE operation. If separate address buses for instructions and data were used, an instruction could be fetched and a data value stored in one clock cycle, thus bringing the average MIPS rates in table 5 to 25 MIPS.

An integral floating point unit could improve system performance by reducing the burden of loading and unloading operands to and from the FPU. Currently, two clock cycles are required to load or unload 32 bit

operands; but with an integral FPU, external accesses could be eliminated by sharing internal registers.

Pipelining is a common technique used to improve the throughput of a CPU; unfortunately, it can cause additional delays when branches are encountered. The optimal CPU for this architecture will not be pipelined in order to avoid the branch problem. The AMD 29027 FPU is also pipelined for maximum execution speed. The data in the FPU must be pushed through the unit with instruction or operand writes until the result is present at its outputs. This fact significantly increases execution time for floating point operands by causing the FPU to take up to 400nS to compute a result when it is capable of performing the same operation in 200nS. For the optimal FPU, pipeline advancement would not be dependent on two cycle LOAD or STORE instructions.

Input/Output Processor

In order for a parallel processing system to operate at peak performance levels, the interprocessor communication structure must be fast and efficient. The I/O processor must be able to retrieve variables from the interconnection bus network and deposit them into the CPUs memory as fast as possible to minimize the communication overhead of the parallel processing system. Although it is beyond the scope of this paper to design an I/O processor, the following paragraphs will attempt to describe a possible I/O processor in sufficient detail to derive realistic times for the communication delays.

Intelligent versus Nonintelligent I/O Processors

There are two basic extremes when it comes to I/O processor design, the intelligent I/O processor and the nonintelligent I/O processor.

The intelligent I/O processor is characterized as having a programmable CPU and the nonintelligent I/O processor is characterized as being a collection of dumb, hardwired state machines.

The intelligent I/O processor has the advantage of being able to handle errors intelligently, perform more complex tasks, and make the system architecture flexible in terms of packet length, content, and format. The nonintelligent or hardwired I/O processor has a speed advantage over a programmable I/O processor since it simply reacts to conditions instead of analyzing them first.

Perhaps the optimal solution is an I/O processor that is a blend of the two extremes. It could contain intelligence in the form of a microprocessor which would monitor the communication process, including the complex scheduling of interprocessor transfers. As long as processes were proceeding normally, the microprocessor would do nothing, and the hardwired data transfer circuitry could proceed at a maximum rate. If an error was detected or some unusual condition occurred, the microprocessor could step in and conduct error recovery or tell the hardware how to handle the unusual condition.

Packet Formats

Now that an I/O processor configuration has been decided upon, a data communication format may be chosen. Figure 10 shows various packet formats that could be used. To simplify the I/O processor design, all packets will use a 32 bit fixed length with one extra bit on the start of the packet to designate what type packet it is, and an extra bit on the end for parity checking. If the prefix bit is a zero, that packet contains a data word; if the prefix bit is a one, that packet contains a command that is directed to one or more PEs.

DATA PACKET

DESCRIPTION	PACKET TYPE ($C/\overline{D}$)	DATA WORD	PARITY
BITS	1	32	1

COMMAND PACKET

DESCRIPTION	PACKET TYPE ($C/\overline{D}$)	CLUSTER ADDRESS	P.E. ADDRESS	COMMAND	PARITY
BITS	1	8	8	16	1

Figure 10. Packet Formats

Interprocessor Communication Times

Using the model of the fiber optic star interconnection network shown in figure 3, the model of the I/O processor discussed in the I/O processor sections, and the model of the communication structure shown in figure 10, an approximation of the time to transfer data from one PE to another can be determined as shown in table 6.

Table 6 gives a total time of 51 clock cycles at the fiber optic star bus frequency and 7 clock cycles at the CPU clock frequency to transfer a data word to an intracluster PE. Using a fiber optic star bus frequency of 300MHz and a CPU frequency of 25MHz, the communication delay time for intracluster transfers will be 450nS. Assuming there is no waiting for access to the intercluster fiber-optic star, no additional delay will be added to the communication delay time for an intracluster transfer.

TABLE 6

INTRACLUSTER COMMUNICATION ANALYSIS

CLOCK CYCLE	FUNCTION PERFORMED
	TRANSMITTING PE
1	PE writes a data word to I/O processor.
2	I/O processor forms a data packet.
3-36	I/O processor shifts data onto the fiber optic star.
	RECEIVING PE
3-36	I/O processor receives data packet and converts to parallel.
37-40	I/O processor computes destination address with look-up table.
41-51	I/O processor performs a read/check/write memory cycle to deposit received data word into PE memory and activates CPU interrupt line.
7 (CPU)	CPU responds to interrupt, writes data word to CPU register location, and sets data received flag.

CHAPTER 5

ARCHITECTURAL EVALUATION

Chapter 5 will attempt to give a better understanding of the performance of this direct-execution parallel architecture by evaluating two ACSL programs in terms of execution speeds and computing resources required.

The Armstrong Cork Benchmark

The Armstrong Cork benchmark, shown in table 2, will now be evaluated to determine the minimum real-time calculation interval possible (Hannaver 1986). The first step in analyzing an ACSL program is to allocate the program constructs to the various PEs and clusters in the system. One such allocation of clusters is shown in figure 11. In order to insure maximum execution speed, equations should be factored for ease of computation. The resulting factored equations that will be integrated every calculation interval are listed below:

```
X1DOT = X1*(-K1-K3*X4-K7*X3)
X2DOT = -K2*X2+X1*(K1+K3*X4+K7*X3)+K9*X4
X3DOT = K6*X5*X5+X3*(-K7*X1-K8*X4)
X4DOT = X4*(-K3*X1-K4*X4-K8*X3-K9)+K2*X2+K6*X5*X5
X5DOT = X5*X5(-K5-K6)+K3*X1*X4
X6DOT = X4*(K4*X4+K8*X3)+K5*X5*X5.
```

Using the allocation of computing tasks shown in figure 11, the

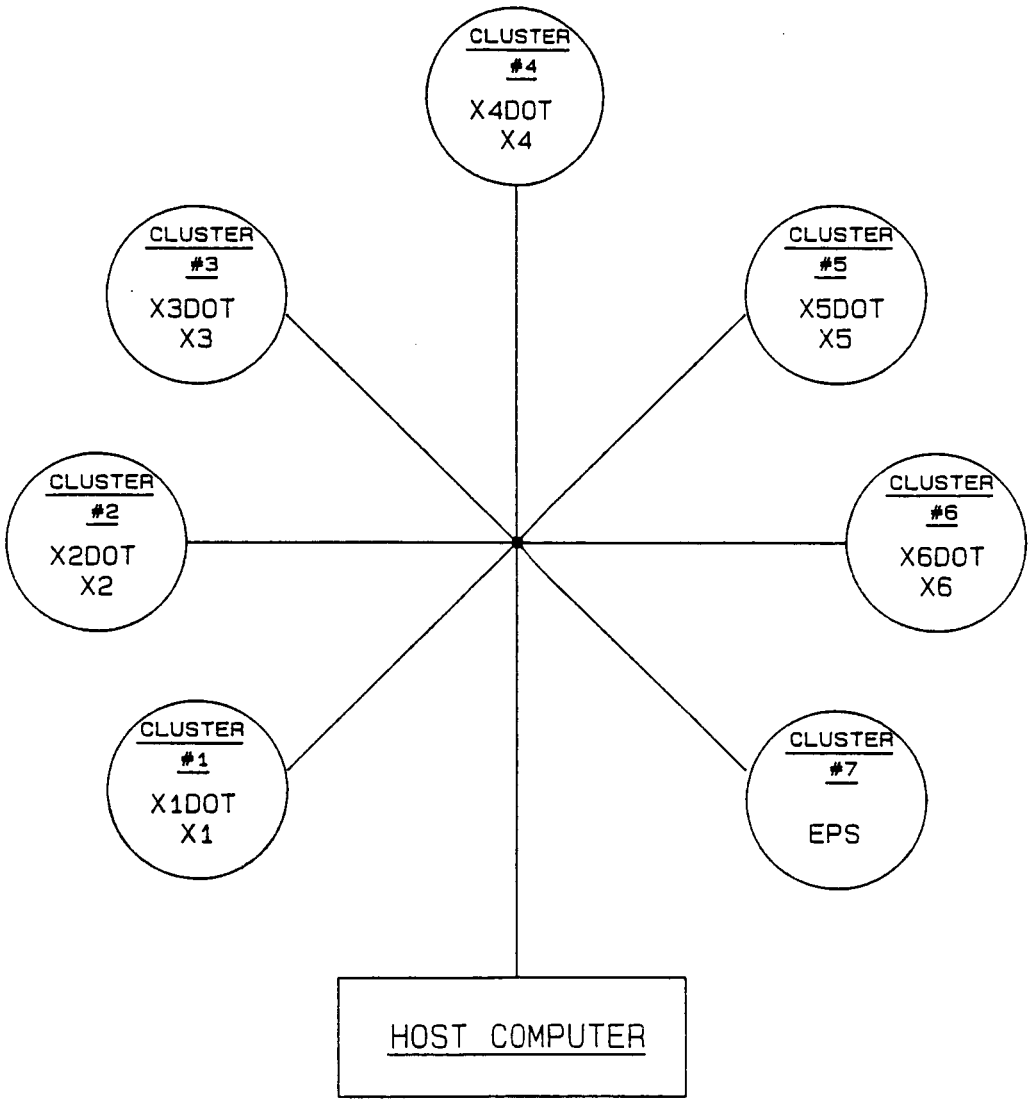


Figure 11. Cluster Allocation for Armstrong Cork Program

calculation times and resources required at each cluster were derived and shown in table 7. The corrector routine of the parallel predictor-corrector method will always take longer to execute than the predictor routine; therefore, only the corrector routine will be analyzed. For the sake of simplicity, it will be assumed that the derivative function is evaluated entirely on one PE. Cluster 7 will be used to evaluate the equation for the output function EPS. The equation for EPS will require five additions which can be evaluated in 1.2uS.

TABLE 7

ARMSTRONG CORK CLUSTER ACTIVITY

CLUSTER	DERIVATIVE EVALUATION TIME	CORRECTOR EVALUATION TIME	PEs USED
Cluster 1	0.880uS	2.49uS	2
Cluster 2	1.6uS	3.21uS	2
Cluster 3	1.52uS	3.13uS	2
Cluster 4	2.24uS	3.85uS	2
Cluster 5	1.2uS	2.81uS	2
Cluster 6	1.44uS	3.05uS	2

To obtain the minimum calculation interval possible with this configuration, the communication delay times of all the state variables must be considered. Since all state variables are used in other equations, their values must be updated every calculation interval. The updating of values will be done after the derivative function is integrated and will result in an additional delay of 1.34uS. The output function EPS will be transmitted to the host computer every calculation

interval for recording. The minimum calculation interval can now be determined by taking the worst execution time for the clusters and adding it to the update delay, 1.34uS. The slowest executing cluster is cluster 4 (3.85uS); therefore, the minimum real-time calculation interval is found to be 5.19uS. This value could possibly be lowered by dividing the function X4DOT among two or more PEs for evaluation. If this method is taken, care should be taken to insure that communication bottlenecks to not occur due to excessive task division.

Dragster Benchmark

The DERIVATIVE portion of an ACSL program named Dragster is shown in table 8 (Hannaver 1986). The Dragster program will be analyzed in the same fashion the Armstrong Cork program was examined. Figure 12 shows several clusters of PEs connected by the intercluster fiber optic star and the various functions allocated to each cluster.

The calculation activity at each cluster will now be explained in detail:

Cluster 1. Cluster 1 is responsible for performing an integration to obtain OMEGAE. The derivative function will require 1.64uS to evaluate, thus making the integration last 3.25uS.

Cluster 2. The integration of OMEGAT will use five PEs, two to perform the predictor-corrector algorithm, one to evaluate the equation for TE, one to evaluate the equation for FRIC, and one to evaluate the equation for FT. The equation for the variable TE will require 0.85uS to evaluate and transmit to the PEs responsible for integration and the equation for FRIC will require 1.65uS to compute and transmit to the PE evaluating FT. The equation for FT will require 690nS to calculate

TABLE 8

DRAGSTER PROGRAM

```
DERIVATIVE

      'THROTTLE CONTROL'
      CONSTANT  TRC = 1.3

      'ENGINE'
      OMEGAE = INTEG((GEAR*OMEGAT - OMEGAE)/TAUE, OMEGE0)
      TE = TRC*TN(OMEGAE)
                           CONSTANT  OMEGE0=100
                           CONSTANT  GEAR = 3.0,    TAUE = .1

      'REAR TIRE'
      IT = 0.5*(MT/GC)*RT**2
      OMEGAT = INTEG((TE - RT*FT)/IT, OMEGT0)
      FT = FRIC*((MB + MT)*G/GC - FF)
                           CONSTANT  OMEGT0=0
                           CONSTANT  MT = 150., RT= 1.8

      'ROAD FRICTION'
      SLIP = (RT*OMEGAT - V)/VMAX
      LS1 = SLIP.GT..15
      LS2 = SLIP.GT..2
      FRIC = RSW( .NOT.LS1, (1.4/.15)*SLIP, 0.) +
             RSW( LS1.AND..NOT.LS2, 1.4, 0.) +
             RSW( LS2, .65, 0.)
                           CONSTANT  VMAX = 500.

      'FORWARD MOTION'
      VDT = (FT-FD)/((MB + MT)/GC)
      V = INTEG( VDT, V0)
      X = INTEG( V, X0)
                           CONSTANT  V0 = 0., X0 = 0.
                           CONSTANT  MB = 1500.

      'AERODYNAMIC DRAG'
      FD = .5*(RHO/GC)*CD*A*V**2
                           CONSTANT  A = 12., CD = .15, RHO = .081

      'BODY ROTATION'
      SINTP = SIN(THETA + PHI)
      COSTP = CON(THETA + PHI)
      IB = (MB/GC)*(1 + LCG**2)
      OMEGAB = INTEG((TE + LWB*FF + (MB/GC)*LCG*VDT*SINTP -
                 (MB*G/GC)*LCG*COSTP)/IB, OMEGB0)
      THETA = INTEG( OMEGAB, THETA0)
                           CONSTANT  OMEGB0=0., THETA0=0.
                           CONSTANT  LCG=4., LWB=18., PHI = .174533

      'FRONT SUSPENSION'
      FF = BOUND(0,5000., -LWB*(KS*THETA + KD*OMEGAB))
```

TABLE 8--Continued

```
                        CONSTANT  KD = 100., KS= 8400.

     'RUN TERMINATION'
     FLIP = THETA.GT.1

     TERMT((TIME.GT.RUNTIM).OR.(X.GT.XMAX).OR.FLIP)

     '@RECORD(REC01,,,,,,,,OMEGAE,VDT,V,X,OMEGAB,THETA, ...
                           OMEGAT,TIME)'

END  $    'DERIVATIVE'
```

after the value for FRIC is received, making the execution time from the start of the calculation interval for evaluating and transmitting FT equal to 2.34uS.

Once TE and FT are computed, the integration of OMEGAT' can continue. The execution time for the derivative expression (measured from the start of the calculation interval) is 3.53uS, thus making the total time necessary to calculate OMEGAT equal to 5.14uS.

Cluster 3. Cluster 3 is responsible for integrating VDT. The equation for VDT can be evaluated on two PEs in 1.48uS, thus making the evaluation time for the integration 3.09uS.

Cluster 4. Cluster 4 is responsible for integrating the variable V. The integration will take 1.61uS.

Cluster 5. Cluster 5 is responsible for computing OMEGAB. The derivative of OMEGAB is composed of FF, SINTP, and COSTP; therefore, values for FF, SINTP, and COSTP will first be evaluated simultaneously on separate PEs to decrease the derivative function evaluation time. FF will require 2.05uS to evaluate and transmit; SINTP will require 1.97uS

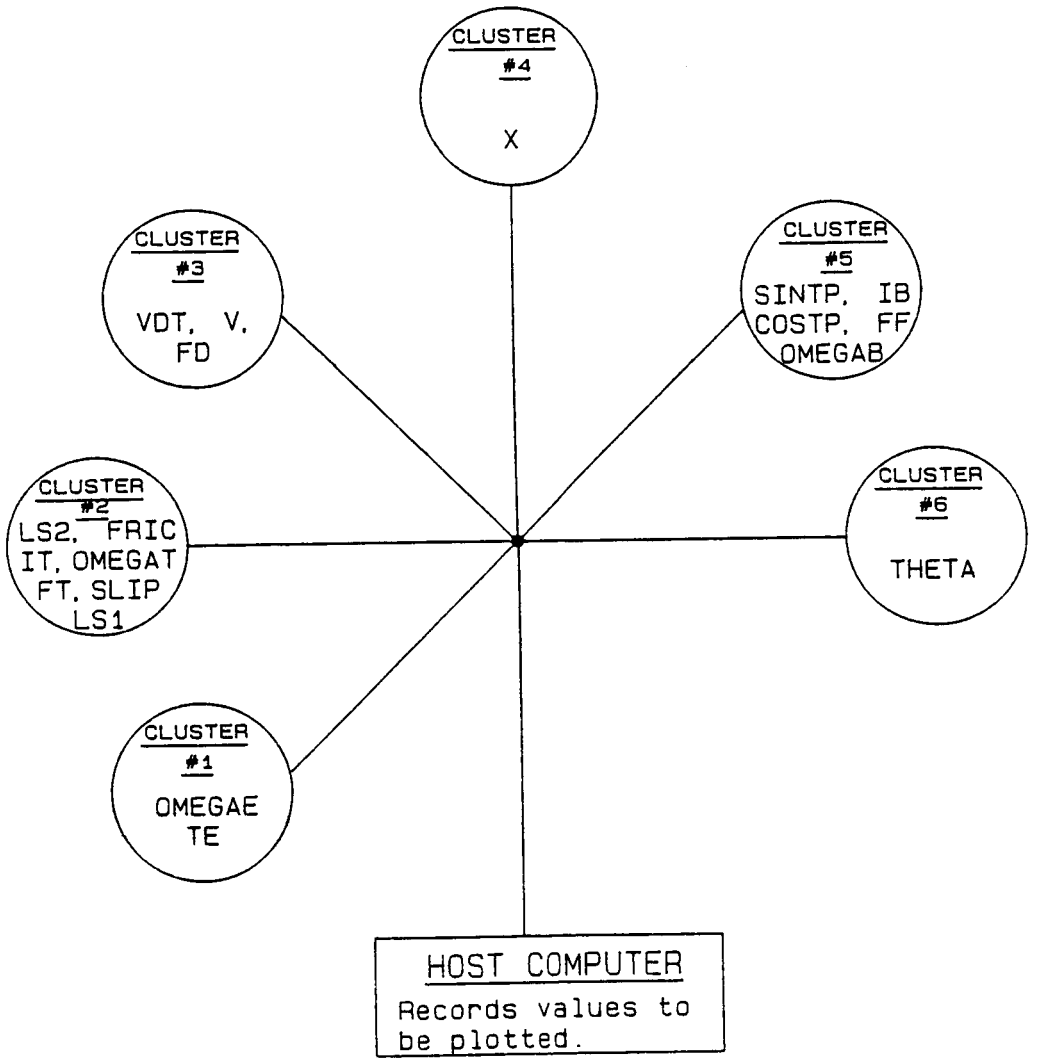


Figure 12. Cluster Allocation for Dragster Program

to calculate and transmit; and COSTP will require 1.81uS to compute and transmit. The function evaluation time for the derivative of OMEGAB will require 2.84uS (after FF, SINTP, and COSTP are computed), bringing the integration time for OMEGAB to 4.45uS. This results in a total cluster execution time of 6.5uS.

Cluster 6. Cluster 6 is responsible for integrating the variable OMEGAB. The execution time for this integration is 1.61uS.

A summary of the cluster execution times and the resources required for the Dragster program is shown in table 9. As in the Armstrong Cork program, the update times must be computed before the maximum calculation interval may be derived. There are 10 variables used by the different clusters of PEs and the host computer (who records values for plotting). When the variables are updated as soon as a new value is calculated, an additional delay of 450nS will result. Adding this delay to the slowest executing cluster (6.5uS) shows the minimum real-time calculation interval is 6.95uS. With further analysis, it is possible that this value could be lowered with a more judicious allocation of processing resources.

TABLE 9

DRAGSTER PROGRAM CLUSTER ACTIVITY

CLUSTER	DERIVATIVE EVALUATION TIME	CORRECTOR EVALUATION TIME	TOTAL CLUSTER TIME	PEs USED
Cluster 1	1.64uS	3.25uS	3.25uS	2
Cluster 2	3.53uS	5.14uS	5.14uS	5
Cluster 3	1.48uS	3.09uS	3.09uS	2
Cluster 4	0	1.61uS	1.61uS	2
Cluster 5	4.89uS	6.50uS	6.50uS	5
Cluster 6	0	1.61uS	1.61uS	2

CHAPTER 6

DISCUSSION OF RESULTS

A direct-execution parallel architecture has been presented that includes an interconnection topology, the requirements for the allocater, a model of a processing element, a model of an I/O processor, the interprocessor communication formats, a survey of current 32 bit RISC microprocessors, a model of an ideal microprocessor, and the microprogramming for the ACSL constructs. Armed with the above items, the execution times and the resources required for two ACSL programs were determined as shown in chapter 5. It should be noted that the execution times derived in chapter 5 are the actual values that should be expected if the architecture was implemented, since all pertinent variables were considered (such as interprocessor communication times and the required data logging).

Parallel versus Serial Execution

To get a better understanding of the results of chapter 5, the execution speeds obtained for the Armstrong Cork program and the Dragster program will now be compared to execution speeds for the same programs obtained from a sequential direct-execution architecture. Assuming that a 25MHz AMD 29000 and an AMD 29027 FPU were being used and that they were executing exclusively from high-speed static RAM with no wait states, then the resulting execution speeds for the Armstrong Cork program and the Dragster program would be the values shown in table 10.

TABLE 10
COMPARISONS BETWEEN SEQUENTIAL AND PARALLEL IMPLEMENTATIONS

PROGRAM	SEQUENTIAL EXECUTION TIMES	PARALLEL EXECUTION TIMES	PERCENT IMPROVEMENT
ARMSTRONG CORK	30.32uS	5.19uS	484%
DRAGSTER	55.12uS	6.95uS	693%

One major difference between the parallel implementation and the sequential implementation is the type of integration routine used. Since there is only one PE in the sequential approach, naturally the parallel predictor-corrector method cannot be used; instead, a serial form of the predictor-corrector method obtained from the Adams pair shown below will be used (Liniger and Miranker 1966):

$$Yp(n+1) = Y(n) + .5h(3Fc(n) - Fc(n-1)) \text{ and}$$

$$Yc(n+1) = Y(n) + .5h(Fp(n+1) + Fc(n)).$$

The time required to compute the above two equations when executed on a single PE is given by:

$$\text{Serial Integration Time} = 1.68uS + 2*(DET).$$

In contrast, the execution time for the parallel corrector algorithm is represented by:

$$\text{Parallel Integration Time} = 1.16uS + DET + CD.$$

DET represents the derivative evaluation time and CD is the communication delay. Comparing the execution time for the parallel predictor-corrector method with the serial case shows that the parallel method

(for a 450nS communication delay) will out-perform the serial method by a factor of 1.04 to 2.0, depending on the derivative evaluation time. These comparisons assume that the derivative evaluation times are the same for both the parallel case and the serial case. This will not be a valid assumption in an optimal parallel system, since a complex derivative function would be divided among several PEs, thus reducing its derivative evaluation time.

Armstrong Cork Program

Summing the individual serial integrations for the Armstrong Cork program results in a total execution time (per calculation period) of 30.32uS. Comparing this value with the parallel case shows that the serial system executes 5.84 times slower than the parallel system. The theoretical maximum increase of 12 (for the derivative evaluated on a single PE) was not realized, because the communication delays necessary to update state variables in the parallel system and the overhead associated with evaluating the predictor-corrector equations were not zero. If the communication delays and the overhead for computing the integration equations are assumed to be zero, or the derivative evaluation times are large enough to make the communication delays and the overhead for the integration equations negligible, then the parallel architecture will operate 12 times faster than the serial architecture.

Dragster Program

The parallel version of the Dragster program had a slightly larger execution speed increase (over the serial version) than the Armstrong Cork program. This was due to the complexity of the derivative functions evaluated and the nature of their equations. Two of the deriva-

tive functions were broken down into smaller parts and computed in parallel, thus giving an additional increase in execution speed. If the communication delays and integration equation evaluation times are assumed to be negligible, the parallel architecture will execute eight times faster than the serial architecture. Since two of the derivative functions do not require any time to calculate, the theoretical maximum speed will be limited to eight times the serial method, not twelve as in the Armstrong Cork program. The theoretical maximum speed ratio assumes that the derivative functions are executed on individual PEs; in order to increase the parallel processing execution times, the allocater could distribute complex derivative functions among several PEs to allow their computation in parallel.

Conclusions

It has been shown that the combination of parallel processing and direct-execution concepts significantly increases execution speeds of ACSL simulations. It appears that the more complex a given ACSL program is, the more benefits parallel processing will provide. It is also apparent that the communication delays have a direct bearing on architecture performance, especially when simple derivative functions are being evaluated. The optimum environment for parallel processing occurs when the derivative functions are large enough to make the communication delays negligible and large enough to allow their restructuring in order to execute portions of them in parallel.

The direct-execution concepts benefit both parallel and sequential architectures by generating the most efficient code possible. The advantages of a direct-execution architecture are highlighted by ACSL programs. The simple, repetitive flow of ACSL programs, along with the

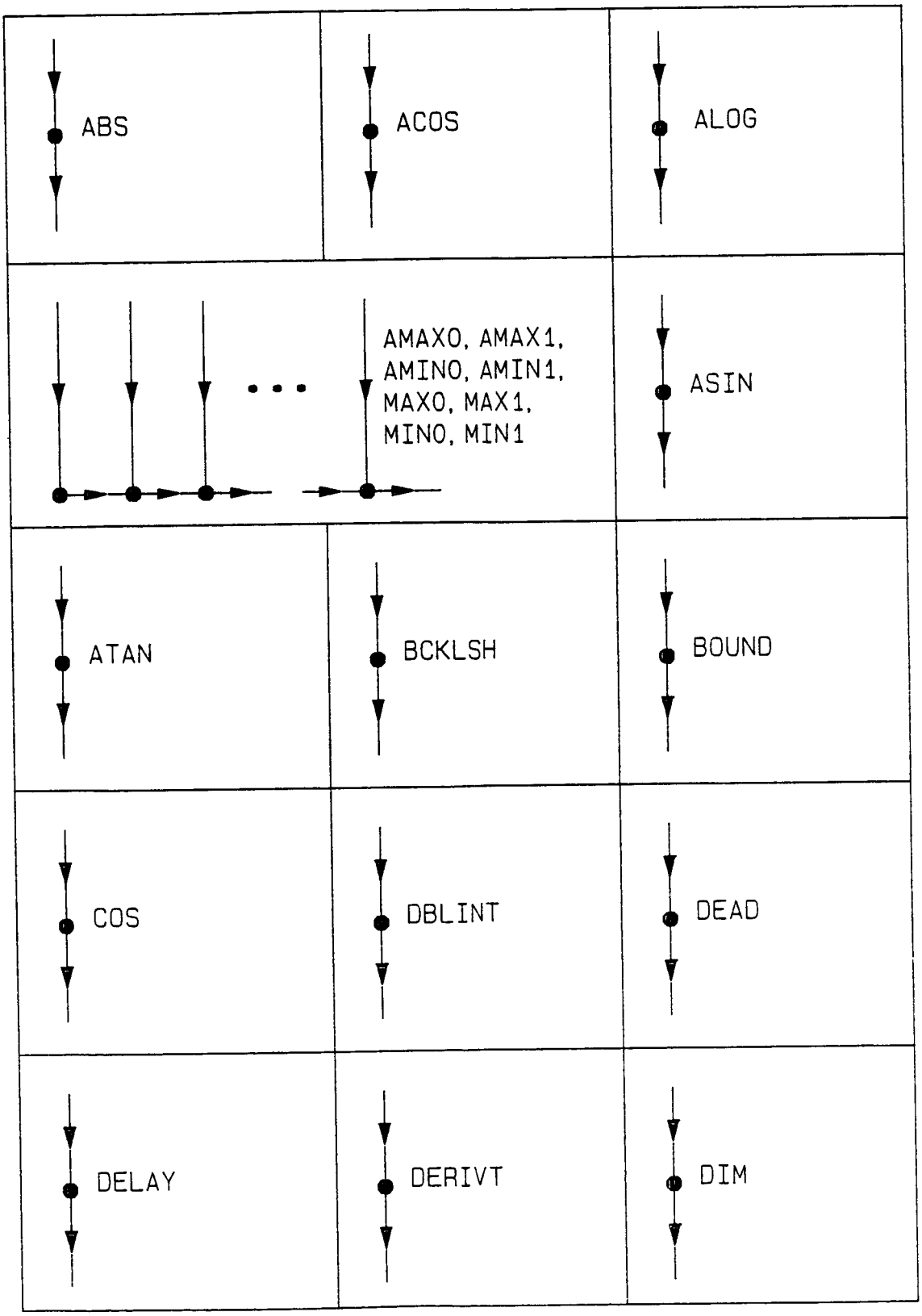
ultra-efficient code generated by the direct-execution approach, allows an ACSL program to be executed to reside in a small block of high-speed static RAM. The small number of operands assigned to individual PEs (due to parallel processing) further enhances performance by allowing variables to be stored in internal CPU registers, rather than slow main memory. By implementing a direct-execution parallel architecture in this manner, performance levels not achievable with sequential computers using compiled code will become a reality.

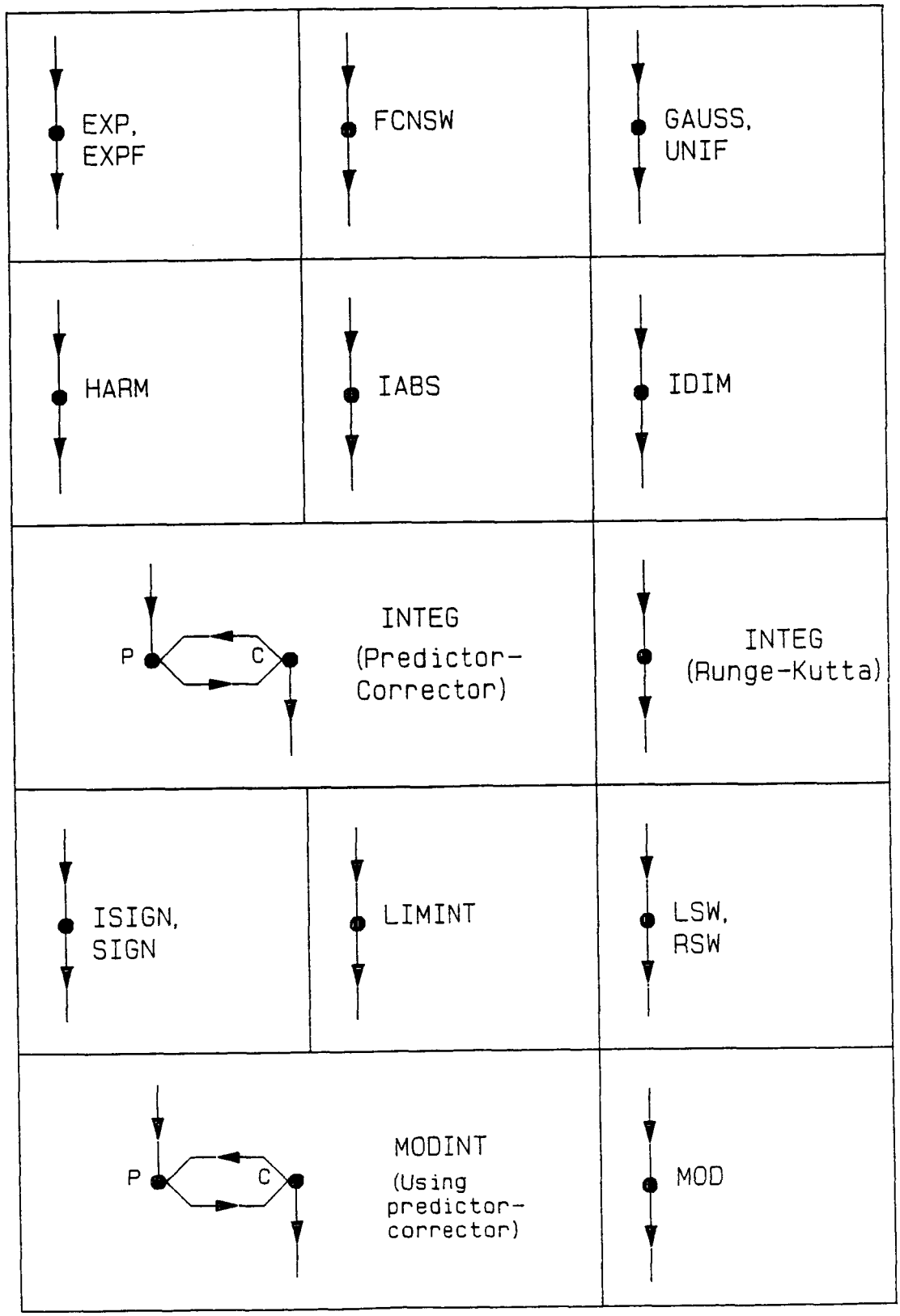
APPENDIX A

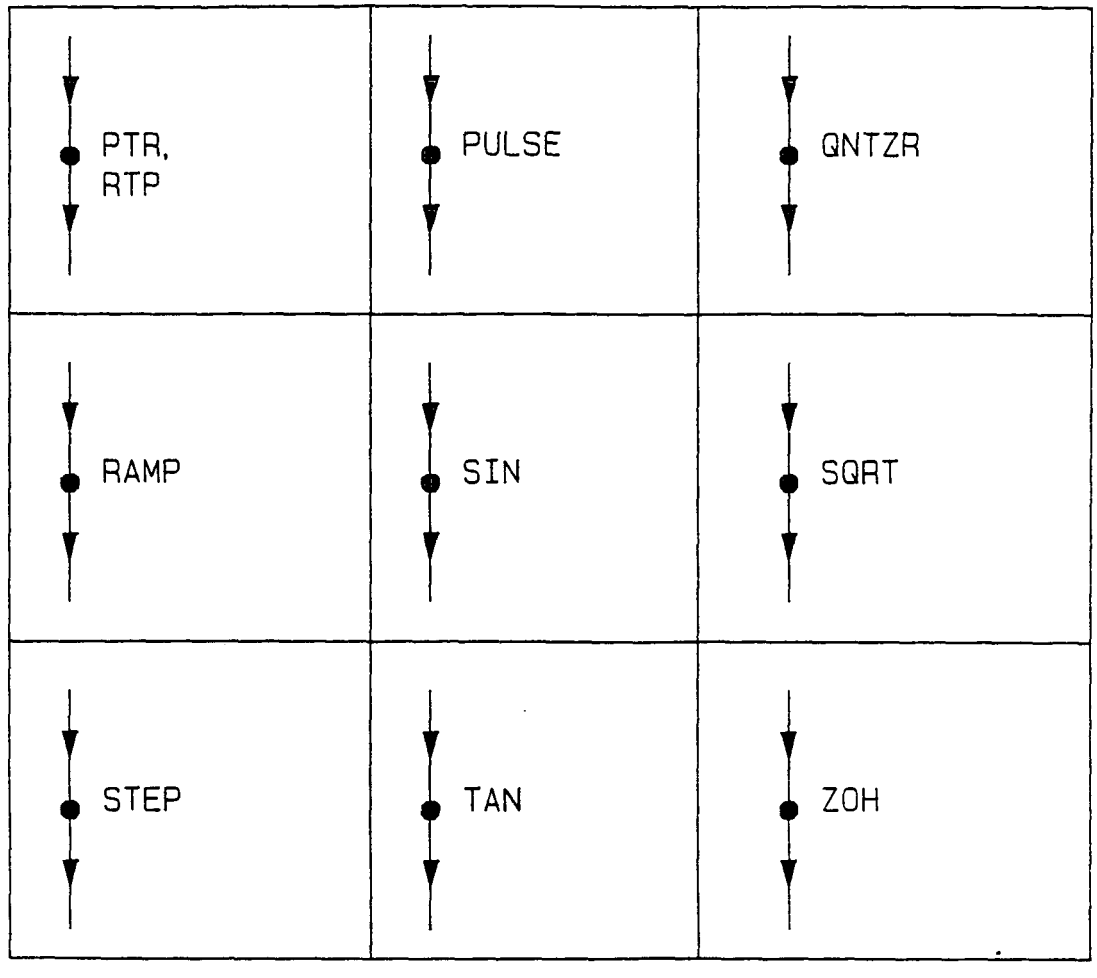
DATA FLOW GRAPHS FOR ACSL CONSTRUCTS

The data flow graphs shown on the following pages represent the data flow between PEs in a parallel processing system. In all the graphs, a vertex represents one PE and the directed arcs represent the direction data flows between PEs.

As illustrated by the graphs, very few of the construct routines contain parallelism. The AMAX, AMIN, MAX, and MIN type constructs contain parallelism such that the optimum number of PEs used to evaluate the functions depends on the number of operands. The integration construct is implemented with a two processor parallel predictor-corrector algorithm. It allows the prediction of the n+1 value while at the same time correcting for the n value.


ABS
ACOS
ALOG
AMAXO, AMAX1,
AMINO, AMIN1,
MAXO, MAX1,
MINO, MIN1
ASIN
ATAN
BCKLSH
BOUND
COS
DBLINT
DEAD
DELAY
DERIVT
DIM


EXP,
EXPF
FCNSW
GAUSS,
UNIF
HARM
IABS
IDIM
P
C
INTEG
(Predictor-
Corrector)
INTEG
(Runge-Kutta)
ISIGN,
SIGN
LIMINT
LSW,
RSW
P
C
MODINT
(Using
predictor-
corrector)
MOD


PTR,
RTP
PULSE
QNTZR
RAMP
SIN
SQRT
STEP
TAN
ZOH

APPENDIX B

MICROPROGRAMMED ROUTINES FOR ACSL CONSTRUCTS

The microprogrammed ACSL routines shown follow standard AMD 29000 assembly language formats, except for the LOAD and STORE instructions needed when loading or storing the FPU. Due to the recent introduction of the AMD 29027 FPU into the marketplace, there was no standard format available pertaining to the programming syntax of coprocessor LOAD or STORE instructions when this thesis was written, so one was devised as follows:

```
STORE FPU INST    PMUX,QMUX,TMUX/INSTRUCTION/REGISTER WRITE
STORE FPU OPT     OPERAND #1, OPERAND #2
STORE FPU OP      OPERAND #1, OPERAND #2
LOAD FPU RES      DESTINATION,RESULT SELECT
```

The AMD 29027 will be operated in a pipelined mode. The three stage pipeline is represented by the three areas in the STORE FPU INST operand field. The first area determines where the ALU operands will come from, the second area determines what operation is performed on the data, and the third area selects the internal FPU register (if any) to deposit results in. This STORE instruction does advance the pipeline.

The STORE FPU OPT indicates what operands to store in the R-TEMP and S-TEMP registers in the FPU. The operand #1 (if any) will be stored in the R-TEMP register, and operand #2 (if any) will be stored in the S-TEMP register. This type of instruction does not advance the pipeline.

The STORE FPU OP indicates what data values to store in the R and S registers of the FPU. This instruction does advance the pipeline and stores data in the same fashion as the STORE FPU OPT instruction, except it deals with the R and S registers rather than the temporary registers.

The LOAD FPU RES instruction reads the F port of the AMD 29027. The data read can be the least significant bits of the result, the most significant bits of the results, the flag register, or the FPU status.

ABS

DESCRIPTION: Absolute value of the argument expression x, where x is a real floating point expression.

EXECUTION TIME (WORST CASE): 40nS
MEMORY WORDS REQUIRED: 1
INPUTS: X
OUTPUTS: Y
CODE:

```
        AND     Y,MSBCLR,X          ;CLEAR THE MSB
```

ACOS

DESCRIPTION: Returns the arc-cosine, ACOS (X), where x is a floating point value between -1.0 and 1.0. Result is a real number in radians between 0 and PI.

EXECUTION TIME (WORST CASE): 1.4uS
MEMORY WORDS REQUIRED: 35
INPUTS: X
OUTPUTS: Y
CODE:

```
        STORE FPU OPT  X,
        STORE FPU INST R,,/-/-
        STORE FPU INST -/P/-
        STORE FPU INST RF0,RF0,/-/RF0 ;STORE X IN RF0
        STORE FPU INST -/P*Q/-
        STORE FPU INST RF1,S,R/-/RF1  ;X SQUARED IN RF1
        STORE FPU OP   A5,A6
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST RF1,RF2,R/-/RF2;ACCUMULATE IN RF2
        STORE FPU OP   A4,
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST RF1,RF2,R/-/RF2
        STORE FPU OP   A3
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST RF1,RF2,R/-/RF2
        STORE FPU OP   A2
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST RF1,RF2,R/-/RF2
        STORE FPU OP   A1
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST RF1,RF2,1/-/RF2
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST -/T+P*Q/-
        STORE FPU INST RF0,RF2,/-/RF2
        STORE FPU INST -/P*Q/-        ;ACCUM * X
        STORE FPU INST R,-,RF2/-/RF2
        STORE FPU OP   PI/2
        STORE FPU INST -/P-T/-        ;PI/2 - SERIES
        STORE FPU INST -/-/F
        LOAD FPU RES   Y,F            ;READ RESULT
```

AINT

DESCRIPTION: Integerize the argument X where X is a floating point value and the output Y is also a floating point value.

EXECUTION TIME (WORST CASE): 280nS
MEMORY WORDS REQUIRED: 7
INPUTS: X
OUTPUTS: Y
CODE:

```
        STORE FPU OPT    X              ;LOAD OPERAND TO FPU
        STORE FPU INST ,,R/-/-          ;CONVERT TO INTEGER
        STORE FPU INST -/INT(T)/-       ;PUSH DATA THROUGH PIPE
        STORE FPU INST ,,RF0/-/RF0
        STORE FPU INST -/FP(T)/-        ;CONVERT TO FLOATING PT.
        STORE FPU INST -/-/F
        LOAD FPU RES    Y, F            ;READ RESULT FROM FPU
```

ALOG

DESCRIPTION: Natural logarithm of real argument X where X is greater than 0.

EXECUTION TIME (WORST CASE): 2.48uS
MEMORY WORDS REQUIRED: 48
INPUTS: X
OUTPUTS: Y
CODE:

```
;COMPUTE (X-1)/(X+1)
          STORE FPU OPT  X,1
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/P-T/-
          STORE FPU INST R,,S/-/RF3
          STORE FPU INST -/P+T/-
          STORE FPU INST RF1,,/-/RF1
;
;PERFORM DIVISION(RF0=RF3/RF1, WITH MUXES SET TO RF0,RF0,/-/RF0)
;DIVISOR = RF1
;DIVIDEND = RF3
;QUOTIENT/RECIPROCAL = RF0
;
          STORE FPU INST -/RECIP_SEED/-
          STORE FPU INST RF0,RF1,2/-/RF0     ;SEED IN RF0
;READY FOR FIRST ITERATION FOR RECIPROCAL DIVISION
;EVALUATE Xi+1 = Xi*(2-B*Xi)
;
AGAIN:    STORE FPU INST -/T-P*Q/-
          STORE FPU INST -/T-P*Q/-
          STORE FPU INST -/-/RF2         ;RF2=2-B*X(i)
          STORE FPU INST RF0,RF2,/-/-
          STORE FPU INST -/P*Q/-
          JMPFDEC   COUNT,AGAIN          ;DO "COUNT" ITERATIONS(3)
          STORE FPU INST RF0,RF1,2/-/RF0
                                         ;RF0=  X(i+1)
          STORE FPU INST RF3,RF0,/-/-    ;MULTIPLY DIVIDEND BY
          STORE FPU INST -/P*Q/-         ;1/DIVISOR.
          STORE FPU INST RF0,RF0,/-/RF0 ;QUOTIENT IS IN RF0 AND F
;
;COMPUTE SERIES FOR ALOG
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF1,S,R/-/RF1  ;Y SQUARED IN RF1
          STORE FPU OP   A5,A6
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2;ACCUMULATE IN RF2
          STORE FPU OP   A4,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A3
          STORE FPU INST -/T+P*Q/-
```

```
STORE FPU INST -/T+P*Q/-
STORE FPU INST RF1,RF2,R/-/RF2
STORE FPU OP   A2
STORE FPU INST -/T+P*Q/-
STORE FPU INST -/T+P*Q/-
STORE FPU INST RF1,RF2,R/-/RF2
STORE FPU OP   A1
STORE FPU INST -/T+P*Q/-
STORE FPU INST -/T+P*Q/-
STORE FPU INST RF1,RF2,1/-/RF2
STORE FPU INST -/T+P*Q/-
STORE FPU INST -/T+P*Q/-
STORE FPU INST RF0,RF2,/-/RF2
STORE FPU INST -/P*Q/-
STORE FPU INST RF2,2,/-/RF2
STORE FPU INST -/P*Q/-
STORE FPU INST -/-/F
LOAD FPU RES   Y,F            ;READ RESULT
```

AMAX0

DESCRIPTION: Determine the maximum argument where the inputs are integers and the output is a floating point value.

EXECUTION TIME (WORST CASE): (7*CNT + 21)*40nS
MEMORY WORDS REQUIRED: 19
INPUTS: J1, J2, J3, ... Jn
OUTPUTS: Y
PARAMETERS:
CNT = NUMBER OF OPERANDS - 1.
IPA = POINTING TO BEGINNING OF STRING
(ASSUME VARIABLES ARE IN GENERAL PURPOSE REGISTERS)
CODE:

```
          OR        Y, IPA, 0
AGAIN:    CPLE      COND, IPA, Y
          JMPT      COND, SKIP          ;IF VALUE < MAX, JUMP
          MFSR      COND,IPAREG
          OR        Y, IPA, 0
SKIP:     ADD       COND,COND,#01       ;POINT TO NEXT VALUE
          JMPFDEC   CNT, AGAIN
          MTSR      IPAREG,COND
;
          STORE FPU OPT   Y             ;CONVERT TO FLOATING PT.
          STORE FPU INST ,,R/-/-
          STORE FPU INST -/FP(T)/-
          STORE FPU INST RF0,,R/-/RF0
;
;WAIT FOR RESULTS FROM OTHER PE
HERE:     JMPF      OPER,HERE
          NOP
;
          STORE FPU OP   X
          STORE FPU INST -/MAX P,T/-
          STORE FPU INST -/-/F
          LOAD FPU INST  Y,F
          STORE     IOP,Y               ;SEND RESULT TO NEXT PE
;
```

AMAX1

DESCRIPTION: Return the maximum argument where the inputs are floating point values and the output is a floating point value.

EXECUTION TIME (WORST CASE): (10*CNT + 21)*40nS
MEMORY WORDS REQUIRED: 19
INPUTS: X1, X2, X3, ... Xn
OUTPUTS: Y
PARAMETERS:
CNT = NUMBER OF OPERANDS - 1
IPA = POINTS TO START OF STRING
(ASSUME ALL OPERANDS ARE IN THE GENERAL PURPOSE REGISTERS)
CODE:

```
          OR         Y, IPA, 0           ;INITIALIZE FPU ACCUM.
          STORE FPU OPT  Y,
          STORE FPU INST R,,/-/-
          STORE FPU INST -/P/-
          STORE FPU INST RF0,,R/-/RF0
;
AGAIN:    STORE FPU OP   IPA             ;LET FPU FIND MAXIMUM
          STORE FPU INST -/MAX P,T/-
          STORE FPU INST RF0,,R/-/RF0    ;STORE NEW MAXIMUM
          MFSR       COND,IPAREG
          ADD        COND,COND,#01
          JMPFDEC    CNT, AGAIN          ;IF NOT COMPARED ALL
          MTSR       IPAREG,COND         ;DO ANOTHER.
;
;WAIT FOR RESULTS FROM OTHER PE
HERE:     JMPF       OPER,HERE
          NOP
;
          STORE FPU OP   X
          STORE FPU INST -/MAX P,T/-
          STORE FPU INST -/-/F
          LOAD FPU INST  Y,F
          STORE      IOP,Y               ;SEND RESULT TO NEXT PE
;
```

AMINO

DESCRIPTION: Determine the minimum argument where the inputs are integers and the output is a floating point value.

EXECUTION TIME (WORST CASE): (7*CNT + 21)*40nS
MEMORY WORDS REQUIRED: 19
INPUTS: J1, J2, J3, ... Jn
OUTPUTS: Y
PARAMETERS:
CNT = NUMBER OF OPERANDS - 1.
IPA = POINTING TO BEGINNING OF STRING
(ASSUME VARIABLES ARE IN GENERAL PURPOSE REGISTERS)
CODE:

```
          OR         Y, IPA, 0
AGAIN:    CPGE       COND, IPA, Y          ;COMPARE CURRENT VALUE
          JMPT       COND, SKIP            ;TO CURRENT MINIMUM.
          MFSR       COND,IPAREG
          OR         Y, IPA, 0
SKIP:     ADD        COND,COND,#01
          JMPFDEC    CNT, AGAIN            ;IF NOT THROUGH, JMP
          MTSR       IPAREG,COND           ;POINT TO NEXT VALUE.

;
          STORE FPU OPT   Y                ;CONVERT MINIMUM VALUE
          STORE FPU INST ,,R/-/-           ;TO FLOATING POINT.
          STORE FPU INST -/FP(T)/-
;
          STORE FPU INST RF0,,R/-/RF0
;
;WAIT FOR RESULTS FROM OTHER PE
HERE:     JMPF       OPER,HERE
          NOP
;
          STORE FPU OP   X
          STORE FPU INST -/MIN P,T/-
          STORE FPU INST -/-/F
          LOAD FPU INST  Y,F
          STORE      IOP,Y                 ;SEND RESULT TO NEXT PE
;
```

AMIN1

DESCRIPTION: Return the minimum argument where the inputs are floating point values and the output is a floating point value.

EXECUTION TIME (WORST CASE): (10*CNT + 21)*40nS
MEMORY WORDS REQUIRED: 19
INPUTS: X1, X2, X3, ... Xn
OUTPUTS: Y
PARAMETERS:
CNT = NUMBER OF OPERANDS - 1
IPA = POINTS TO START OF STRING
(ASSUME ALL OPERANDS ARE IN THE GENERAL PURPOSE REGISTERS)
CODE:

```
          OR        Y, IPA, 0          ;INITIALIZE FPU ACCUM.
          STORE FPU OPT  Y,
          STORE FPU INST R,,/-/-
          STORE FPU INST -/P/-
          STORE FPU INST RF0,,R/-/RF0
;
AGAIN:    STORE FPU OP   IPA            ;LET FPU FIND MINIMUM.
          STORE FPU INST -/MIN P,T/-
          STORE FPU INST RF0,,R/-/RF0   ;STORE NEW MINIMUM
          MFSR      COND,IPAREG
          ADD       COND,COND,#01
          JMPFDEC   CNT, AGAIN          ;IF NOT COMPARED ALL
         MTSR      IPAREG,COND         ;DO ANOTHER.
;
;WAIT FOR RESULTS FROM OTHER PE
HERE:     JMPF      OPER,HERE
          NOP
;
          STORE FPU OP   X
          STORE FPU INST -/MIN P,T/-
          STORE FPU INST -/-/F
          LOAD FPU INST  Y,F
          STORE     IOP,Y               ;SEND RESULT TO NEXT PE
;
```

AMOD

DESCRIPTION: Remainder of modulus, AMOD(X1,X2), where the floating point remainder of X1 divided by X2 is returned.

EXECUTION TIME (WORST CASE): 1.6uS
MEMORY WORDS REQUIRED: 26
INPUTS: X1, X2
OUTPUTS: Y
CODE:

```
        MACRO     FDIV(RF0,X1,X2)      ;DIVIDE X1 BY X2 AND
                                       ;RETURN RESULT IN RF0.
        STORE FPU INST -/ROUND T/-     ;ROUND RESULT TO LOWER
        STORE FPU INST R,RF0,/-/RF0    ;WHOLE NUMBER.
        STORE FPU OP   X2              ;MULTIPLY ROUNDED RESULT
        STORE FPU INST -/P*Q/-         ;WITH 1/DIVISOR.
        STORE FPU INST R,,RF0/-/RF0
        STORE FPU OP   X2,             ;SUBTRACT PRODUCT FROM
        STORE FPU INST -/P-T/-         ;DIVIDEND.
        STORE FPU INST -/-/F
        LOAD FPU RES   Y,F             ;READ THE REMAINDER.
```

ASIN

DESCRIPTION: The arc-sine of the real argument X is returned where x is between -1.0 and 1.0, and the result is between -PI/2 and PI/2.

EXECUTION TIME (WORST CASE): 1.24uS
MEMORY WORDS REQUIRED: 31
INPUTS: X
OUTPUTS: Y
CODE:

```
          STORE FPU OPT  X,
          STORE FPU INST -/P/-
          STORE FPU INST RF0,RF0,/-/RF0 ;X IN RF0
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF1,S,R/-/RF1  ;X SQUARED IN RF1
          STORE FPU OP   A5,A6
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A4,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A3
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,1/-/RF2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF2,/-/RF2
          STORE FPU INST -/P*Q/-        ;ACCUM * X
          STORE FPU INST -/-/F
          LOAD FPU RES   Y,F
```

ATAN

DESCRIPTION: Returns the arc-tangent of the real value X.

EXECUTION TIME (WORST CASE): 3.08uS
MEMORY WORDS REQUIRED: 104
INPUTS: X
OUTPUTS: Y
CODE:

```
;CHECK FOR X>1
          STORE FPU OPT  X,1
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/FLAG
          LOAD FPU RES   COMPREG,FLAG
;CHECK > FLAG
          AND       COMPREG,COMPREG,#10H
          CPEQ      COMPTEST,COMPREG,#10H
          JMPT      COMPTEST, XOFR
          STORE FPU OPT  X,-1
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/FLAG
          LOAD FPU RES   COMPREG,FLAG
;CHECK < FLAG
          AND       COMPREG,COMPREG,#08H
          CPEQ      COMPTEST,COMPREG,#08H
          JMPT      COMPTEST,XOFR
;X IS BETWEEN -1 AND +1
          STORE FPU OPT  X,
          STORE FPU INST R,,/-/-
          STORE FPU INST -/P/-
          STORE FPU INST RF0,RF0,/-/RF0 ;X IN RF0
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF1,S,R/-/RF1  ;X SQUARED IN RF1
          STORE FPU OP   A5,A6
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A4,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A3
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A1
          STORE FPU INST -/T+P*Q/-
```

```
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST RF1,RF2,1/-/RF2
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST RF0,RF2,/-/RF2
            STORE FPU INST -/P*Q/-          ;ACCUM * X
            STORE FPU INST -/-/F
            LOAD FPU RES   Y,F
            JMP        END
            NOP
XOFR:
;           COMPUTE ((((((B7)Y+B5)Y+B4)Y+B3)Y+B2)Y+B1)Y+1)Z
;           WHERE Y = 1/(X*X) AND Z = 1/X.
;
            MACRO      RF0=RECIP(X)         ;PUT 1/X IN RF0
;
            STORE FPU INST RF0,RF0,/-/-
            STORE FPU INST -/P*Q/-
            STORE FPU INST RF1,S,R/-/RF1   ;PUT 1/(X*X) IN RF1
;
            STORE FPU OP   B5,B6           ;COMPUTE SERIES
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST RF1,RF2,R/-/RF2
            STORE FPU OP   B4,
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST RF1,RF2,R/-/RF2
            STORE FPU OP   B3
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST RF1,RF2,R/-/RF2
            STORE FPU OP   B2
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST RF1,RF2,R/-/RF2
            STORE FPU OP   B1
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST RF1,RF2,1/-/RF2
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST -/T+P*Q/-
            STORE FPU INST RF0,RF2,/-/RF2
            STORE FPU INST -/P*Q/-          ;ACCUM * X
            STORE FPU INST R,,S/-/RF0       ;RESULT IN RF0
            STORE FPU OP   X,1
            STORE FPU INST -/COMPARE P,T/-
            STORE FPU INST R,,RF0/-/F       ;SEE IF X > 1
            LOAD FPU RES   COMP,FLAG
            AND        COMP,COMP,#10H       ;CHECK > FLAG
            CPEQ       COMP,COMP,#10H
            JMPT       COMP,SKIPNEG         ;IF X > 1, JMP
            NOP
            OR         PIO2,NEGATE,PIO2     ;MAKE PI/2 NEGATIVE
```

```
SKIPNEG:  STORE FPU OP    PIO2,
          STORE FPU INST -/P-T/-
          STORE FPU INST -/-/F
          LOAD FPU RES    Y,F              ;READ RESULT
END:
```

BCKLSH

DESCRIPTION: Used to implement the backlash or hysteresis operator.

EXECUTION TIME (WORST CASE): 960nS
MEMORY WORDS REQUIRED: 28
INPUTS: X
OUTPUTS: Y
PARAMETERS:
2DL = WIDTH OF BACKLASH
IC = INITIAL CONDITION ON THE OUTPUT.
CODE:

```
          STORE FPU OPT   X,Y           ;COMPUTE X - Y
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/P-T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   DIFF,F         ;READ RESULT
          AND       TEMP1, DIFF, CONST1 ;REM STATUS ABOUT X-Y
          AND       DIFF, DIFF, CONST2  ;TAKE ABS OF DIFFERENCE
          STORE FPU OPT  DIFF, 2DL      ;COMPARE DIFF TO WIDTH.
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/FLAG
          LOAD FPU RES   STATUS, FLAG
          AND       STATUS, STATUS, 10H ; CHECK GREATER THAN FLAG
          CPEQ      COND, STATUS, 10H   ; IF WITHIN WIDTH, QUIT
          JMPF      COND, END
          NOP
;INPUT/OUTPUT DIFFERENCE OUT OF RANGE, SO ADJUST OUTPUT
          JMPF      TEMP1, POS          ;CHECK IF + OR - DIFF.
          STORE FPU OPT   X,2DL         ;IN EITHER CASE, LOAD FPU
          STORE FPU INST R,,S/-/-       ;COMPUTE X + 2DL
          STORE FPU INST -/P+T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   Y,F
          JMP       END
          NOP
;
POS:      STORE FPU INST R,,S/-/-       ;COMPUTE X - 2DL
          STORE FPU INST -/P-T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   Y,F
;
END:
```

BOUND

DESCRIPTION: The bound function is used to limit a variable to a particular range.

EXECUTION TIME (WORST CASE): 800nS
MEMORY WORDS REQUIRED: 23
INPUTS: X
OUTPUTS: Y
PARAMETERS:
LL = LOWER LIMIT
UL = UPPER LIMIT
CODE:

```
          OR        Y, X, #00          ;ASSUME IN PROPER RANGE
          STORE FPU OPT   X, LL        ;COMPARE X AND LL
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/FLAG
          LOAD FPU RES   COMP, FLAG
          AND       COMP, COMP, 10H    ;CHECK > FLAG
          CPEQ      COMP, COMP, 10H
          JMPT      COMP, SKIPIT
          NOP
          OR        Y, LL, 00          ;SET OUTPUT TO LL, QUIT
          JMP       END
          NOP
;
SKIPIT:   STORE FPU OPT X,UL           ;COMPARE X AND UL
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/FLAG
          LOAD FPU RES COMP,FLAG
          AND       COMP, COMP, 10H    ;CHECK > FLAG
          CPEQ      COMP, COMP, 10H
          JMPF      COMP, END
          NOP
          OR        Y, UL, 00          ;SET OUTPUT TO UL
END:
```

COS

DESCRIPTION: Returns the cosine of the argument X where the result will be between -1.0 and 1.0 and the argument is in radians.

EXECUTION TIME (WORST CASE): 1.12uS
MEMORY WORDS REQUIRED: 28
INPUTS: X
OUTPUTS: Y
CODE:

```
          STORE FPU OPT  X,
          STORE FPU INST R,R,/-/-
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF1,S,R/-/RF1  ;X SQUARED IN RF1.
          STORE FPU OP   A5,A6          ;COMPUTE SERIES.
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2;ACCUMULATE IN RF2.
          STORE FPU OP   A4,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A3
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,1/-/RF2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/-/F
          LOAD FPU RES   Y,F            ;READ RESULT
```

DBLINT

DESCRIPTION: Provided to limit the second integral of an acceleration (displacement).

EXECUTION TIME (WORST CASE): 720nS
MEMORY WORDS REQUIRED: 22
INPUTS: XDD (ACCELERATION)
OUTPUTS: X (DISPLACEMENT), XD (VELOCITY)
PARAMETERS:
XIC = INITIAL CONDITION ON DISPLACEMENT
XDIC = VELOCITY INITIAL CONDITION
LL = LOWER DISPLACEMENT LIMIT
UL = UPPER DISPLACEMENT LIMIT
CODE:

(TO BE INSERTED AT THE END OF THE INTEGRATION ROUTINE)

```
          STORE FPU OPT   X,UL
          STORE FPU INST R,,S/-/-        ;FIND MAXIMUM OF X AND UL
          STORE FPU INST -/MAX P,T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   GPR1,F          ;READ RESULT OF OPERATION
          CPEQ      COND, GPR1, X        ;SEE WHICH GREATER
          JMPF      COND, SKIP           ;IF X<UL, SKIP
          NOP
          OR         X,UL,00             ;MOVE UL TO X
          AND       XD,XD,00             ;MAKE VELOCITY = 0
          JMP        END
          NOP
SKIP:     STORE FPU OPT  X,LL
          STORE FPU INST R,,S/-/-        ;FIND MINIMUM BTWN X, LL
          STORE FPU INST -/MIN P,T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   GPR1,F
          CPEQ      COND, GPR1, X        ;SEE IF X IS MINIMUM
          JMPF      END                  ;IF NOT, SKIP
          NOP
          OR         X,LL,00             ;MAKE OUTPUT LL
          AND        XD,XD,00            ;MAKE VELOCITY 00
END:
```

DEAD

DESCRIPTION: Used to create dead space in a system. If X is between limits, output is zero.

EXECUTION TIME (WORST CASE): 840nS
MEMORY WORDS REQUIRED: 29
INPUTS: X
OUTPUTS: Y
PARAMETERS:
LL = LOWER LIMIT
UL = UPPER LIMIT
CODE:

```
          STORE FPU OPT  X,UL
          STORE FPU INST R,,S/-/-        ;FIND THE GREATER VALUE
          STORE FPU INST -/MAX P,T/-
          STORE FPU INST -/-/F
          LOAD FPU OP    GPR,F           ;READ RESULT
          CPEQ      COND,GPR,X           ;SEE IF X IS GREATER
          JMPT      COND,CVRUL           ;IF X > UL, JMP
          NOP
;NOW CHECK TO SEE IF X IS LESS THAN LL
          STORE FPU OPT  X,LL
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/MIN P,T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   GPR, F          ;READ RESULT FROM FPU
          CPEQ      COND, GPR, X         ;SEE IF X LESS THAN LL
          JMPT      COND,UNDLL
          NOP
;DEAD SPACE
          JMP       END
          AND       Y,Y,00               ;MAKE OUTPUT 0
;
OVRUL:    STORE FPU OPT  X,UL
          STORE FPU INST R,,S/-/-        ;CALCULATE X - UL
          STORE FPU INST -/P-T/-
          STORE FPU INST -/-/F
          JMP       END
          LOAD FPU RES   Y, F            ;READ RESULT INTO OUTPUT
;
UNDLL:    STORE FPU OPT  X,LL
          STORE FPU INST R,,S/-/-        ;COMPUTE X - LL
          STORE FPU INST -/P-T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   Y, F            ;READ RESULT INTO OUTPUT
;
END:
```

DELAY

DESCRIPTION: Used to model delays through such objects as pipes. A 2*NMX long array is created to model the delay and is written in a circular fashion. It is initially filled with the value IC. The pointer to the output values is set a fixed length from the pointer to the input values during the preprocessing stage to represent the appropriate delay period.

EXECUTION TIME (WORST CASE): 480nS
MEMORY WORDS REQUIRED: 12
INPUTS: X
OUTPUTS: Y
PARAMETERS:
IC - INITIAL CONDITION OF OUTPUT UNTIL FIRST DELAY PERIOD.
TDL - THE DELAY BETWEEN THE INPUT AND THE OUTPUT.
NMX - A CONSTANT REPRESENTING THE NUMBER OF CALCULATION INTERVALS IN THE DELAY.
START - STARTING ADDRESS OF TABLE.
MAXPTR - LAST ADDRESS IN TABLE.
CODE:

```
          STORE     X,INPPTR              ;STORE NEW INPUT
          ADD       INPPTR,INPPTR,#01     ;POINT TO NEXT INPUT
          CPEQ      COND,INPPTR,MAXPTR    ;SEE IF AT END OF TABLE
          JMPF      COND, SKIPCLR         ;DON'T RESET IF NOT
          NOP
          OR        INPPTR,START,#00      ;RESET STARTING ADDRESS
SKIPCLR:  LOAD      Y,OUTPTR              ;READ NEW OUTPUT VALUE
          ADD       OUTPTR,OUTPTR,#01     ;POINT TO NEW OUTPUT
          CPEQ      COND,OUTPTR,MAXPTR    ;SEE IF AT END
          JMPF      COND,SKIP
          NOP
          OR        OUTPTR,START,#00      ;RESET OUTPUT POINTER
SKIP:
```

DERIVT

DESCRIPTION: Implements a first order derivative function in the form:

$$y = (Xnew - Xold)/(Tnew - Told)$$

EXECUTION TIME (WORST CASE): 1.48uS
MEMORY WORDS REQUIRED: 23
INPUTS: X
OUTPUTS: Y
CODE:

```
;  COMPUTE XNEW - XOLD
          STORE FPU OPT  X,XOLD
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/P-T/-
          STORE FPU INST R,,S/-/RF7
;  COMPUTE TNEW - TOLD
          STORE FPU OP   T,TOLD
          STORE FPU INST -/P-T/-
          STORE FPU INST -/-/RF6
;  COMPUTE X/T
          MACRO     F=FDIV(RF7,RF6)
;
          LOAD FPU RES   Y,F            ;READ ANSWER FORM FPU
          OR        TOLD,T,00           ;UPDATE OLD TIME VALUE
          OR        XOLD,X,00           ;UPDATE OLD X VALUE
```

DIM

DESCRIPTION: Positive difference function, DIM(X1,X2). If X1 is greater than X2, returns X1-X2, otherwise returns 0.

EXECUTION TIME (WORST CASE): 400nS
MEMORY WORDS REQUIRED: 10
INPUTS: X1, X2
OUTPUTS: Y
CODE:

```
          STORE FPU OPT  X1,X2          ;COMPUTE X1-X2 AND COMP.
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   COMP,FLAG
          AND       COMP,COMP,#10       ;CHECK G.T. FLAG
          CPEQ      COMP,COMP,#10       ;
          JMPT      COMP,END            ;IF X1 GT X2, JMP
          LOAD FPU RES   Y,F            ;READ X1-X2
          AND       Y,Y,0               ;CLEAR OUTPUT
END:
```

EXP

DESCRIPTION: Returns the natural exponential of the argument.

EXECUTION TIME (WORST CASE): 1.24uS
MEMORY WORDS REQUIRED: 31
INPUTS: X
OUTPUTS: Y
CODE:

```
;IMPLEMENT THE SERIES:
;EXP(X)=1+X(1+X(A0+X(A1+X(A2+X(A3+X(A4+X(A5)))))))
;
          STORE FPU OPT  X,              ;PUT X IN RF0
          STORE FPU INST R,,/-/-         ;ACCUMULATE IN RF1
          STORE FPU INST -/P/-
          STORE FPU INST RF0,R,S/-/RF0
          STORE FPU OP   A5,A4
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,R/-/RF1
          STORE FPU OP   A3,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,R/-/RF1
          STORE FPU OP   A2,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,R/-/RF1
          STORE FPU OP   A1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,R/-/RF1
          STORE FPU OP   A0,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,1/-/RF1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,1/-/RF1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/-/F
          LOAD FPU RES   Y,F             ;READ RESULT
```

EXPF

DESCRIPTION: Implements a switchable exponential depending on the constant ON. If ON is true, a rising exponential from zero to 1.0 is created, and if ON is false, a decaying exponential from 1.0 to zero is implemented.

EXECUTION TIME (WORST CASE): 1.56uS
MEMORY WORDS REQUIRED: 39
INPUTS: ON - SWITCH FUNCTION
OUTPUTS: Y
PARAMETERS:
TA - TIME CONSTANT
IC - Y(0)
T0 - TIME VALUE CORRESPONDING TO Y(0). [EVALUATED IN THE PREPROCESSING STAGE]
T - CURRENT TIME VALUE
CODE:

```
;  EVALUATE EXP[-TA*T]

          STORE FPU OPT  T,T0
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/P+T/-          ;CALCULATE T + T0
          STORE FPU INST R,RF0,/-/RF0
          STORE FPU OP   TA,
          STORE FPU INST -/(-P)*Q/-       ;CALCULATE -TA*(T + T0)
          STORE FPU INST RF0,R,S/-/RF0
;
[EVALUATE EXP(RF0)]
          STORE FPU OP   A5,A4            ;EVALUATE EXP SERIES
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,R/-/RF1
          STORE FPU OP   A3,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,R/-/RF1
          STORE FPU OP   A2,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,R/-/RF1
          STORE FPU OP   A1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,R/-/RF1
          STORE FPU OP   A0,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,1/-/RF1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF1,1/-/RF1
          STORE FPU INST -/T+P*Q/-
```

```
        JMPF      ON,OFF              ;IF OFF, SKIP
        STORE FPU INST -/T+P*Q/-
;
        STORE FPU INST 1,-,RF0/-/RF0
        STORE FPU INST -/P-T/-
        STORE FPU INST -/-/F
        LOAD  FPU RES  Y,F            ;OUTPUT 1-EXP(X)
        JMP       END
        NOP
OFF:    STORE FPU INST RF0,,/-/-
        STORE FPU INST -/P/-
        STORE FPU INST -/-/F
        LOAD FPU RES   Y,F            ;OUTPUT EXP(X)
END:
```

FCNSW

DESCRIPTION: Implements a functional switch where:

```
Y = X1    IF   P < 0,
Y = X2    IF   P = 0, and
Y = X3    IF   P > 0.
```

EXECUTION TIME (WORST CASE): 600nS
MEMORY WORDS REQUIRED: 18
INPUTS: P
OUTPUTS: Y
CODE:

```
          STORE FPU OPT  P,-              ;COMPARE P TO 0
          STORE FPU INST R,,0/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/FLAG
          LOAD FPU RES   CHK,FLAG
          AND       CHK1,CHK,#20H         ;CHECK '=' FLAG
          CPEQ      CHK1,CHK1,#20H
          JMPF      CHK1,NEXT
          OR        Y,X2,00
          JMP       END
          NOP
NEXT:     AND       CHK1,CHK,#10H         ;CHECK '>' FLAG
          CPEQ      CHK1,CHK1,#10H        ;
          JMPF      CHK1,NEXT1
          OR        Y,X3,00               ;MAKE OUTPUT X3
          JMP       END
          NOP
NEXT1:    OR        Y,X1,00               ;ASSUME <
END:
```

GAUSS

DESCRIPTION: Generates a normally distributed random variable with mean M and standard deviation S.

EXECUTION TIME (WORST CASE): 9.8uS
INSTRUCTIONS EXECUTED (WORST CASE): 153
MEMORY WORDS REQUIRED: 21
INPUTS: NONE
OUTPUTS: Y
PARAMETERS:
M - MEAN
S - STANDARD DEVIATION
CODE:

```
;  Y = M + S*Z  WHERE Z = SUMMATION (K=1 TO 12) OF N(K) - 6.
;  N(K) IS A RANDOM NUMBER BETWEEN 0 AND 1.
;
          STORE FPU INST 0,,/-/-        ;INITIALIZE RF1 TO 0
          STORE FPU INST -/P/-
          STORE FPU INST R,,S/-/RF1     ;INIT. ACCUM. REG.
          OR        COUNT,ZERO,#012D    ;INITIALIZE COUNT REG.
AGAIN:    LOAD      N,RDNPTR            ;READ NEW RANDOM NUMBER
          ADD       RDNPTR,RDNPTR,#01   ;POINT TO NEW R.N.
          CPEQ      COND,RDNPTR,MAXPTR  ;SEE IF AT END
          JMPF      COND,SKIP
          NOP
          OR        RDNPTR,START,#00    ;RESET OUTPUT POINTER
SKIP:     STORE FPU OPT  N,SIX          ;COMPUTE N-6
          STORE FPU INST -/P-T/-
          STORE FPU INST RF0,,RF1/-/RF0 ;STORE N-6
          STORE FPU INST -/P+T/-        ;ACCUMULATE VALUES
          JMPFDEC   COUNT,AGAIN         ;IF NOT DONE 12 TERMS,
                                        ;DO ANOTHER.
          STORE FPU INST R,,S/-/RF1
;
;NOW COMPUTE Y = M + S*RF1
;
          STORE FPU OP   S,M            ;STORE MEAN AND S.D.
          STORE FPU INST -/P*Q+T/-
          STORE FPU INST -/P*Q+T/-      ;COMPUTE S*Z+M
          STORE FPU INST -/-/F
          LOAD FPU RES   Y,F            ;READ ANSWER
```

HARM

DESCRIPTION: A sinusoid drive function can be created by this instruction which results in the following:

```
y = 0.0          t < tz,
y = SIN[w*(t-tz) + p].
```

EXECUTION TIME (WORST CASE): 1.88uS
MEMORY WORDS REQUIRED: 47
INPUTS: NONE
OUTPUTS: Y
PARAMETERS:
TZ - DELAY IN SECONDS
W - FREQUENCY IN RAD/SEC
P - PHASE SHIFT IN RADIANS
T - CURRENT TIME
CODE:

```
          STORE FPU OPT  TZ,T              ;COMPARE TIME TO DELAY
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/FLAG
          LOAD FPU RES   CHK,FLAG_REGISTER
          AND       CHK,CHK,#10H          ;CHECK GREATER THAN FLAG
          CPEQ      CHK,CHK,#10H
          JMPT      END
          AND       Y,X,#00               ;CLEAR OUTPUT
;COMPUTE THE SINE FUNCTION
          STORE FPU OPT  T,TZ
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/P-T/-
          STORE FPU INST R,RF0,/-/RF0
          STORE FPU OP   W,
          STORE FPU INST -/P-T/-
          STORE FPU INST RF0,,R/-/RF0
          STORE FPU OP   P,
          STORE FPU INST -/P+T/-
;SINE ROUTINE, X IN RF0
          STORE FPU INST RF0,RF0,/-/RF0 ;X IN RF0
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF1,S,R/-/RF1  ;X SQUARED IN RF1
          STORE FPU OP   A5,A6
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A4,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A3
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
```

```
STORE FPU OP   A2
STORE FPU INST -/T+P*Q/-
STORE FPU INST -/T+P*Q/-
STORE FPU INST RF1,RF2,R/-/RF2
STORE FPU OP   A1
STORE FPU INST -/T+P*Q/-
STORE FPU INST -/T+P*Q/-
STORE FPU INST RF1,RF2,1/-/RF2
STORE FPU INST -/T+P*Q/-
STORE FPU INST -/T+P*Q/-
STORE FPU INST RF0,RF2,/-/RF2
STORE FPU INST -/P*Q/-         ;ACCUM * X
STORE FPU INST -/-/F
LOAD FPU RES   Y,F
```

IABS

DESCRIPTION: Returns absolute value of an integer.

EXECUTION TIME (WORST CASE): 200nS
MEMORY WORDS REQUIRED: 5
INPUTS: J
OUTPUTS: N
CODE:

```
          STORE FPU OPT  J,              ;LET THE FPU COMPUTE
          STORE FPU INST R,,/-/-         ;THE ABSOLUTE VALUE
          STORE FPU INST -/IABS(P)/-     ;OF THE 2'S COMP. INT.
          STORE FPU INST -/-/F
          LOAD FPU RES   N,F
```

IDIM

DESCRIPTION: Returns integer positive difference when:

N = J1 - J2	J1>J2
N = 0	J2>J1.

EXECUTION TIME (WORST CASE): 160nS
MEMORY WORDS REQUIRED: 4
INPUTS: J1, J2
OUTPUTS: N
CODE:

```
          SUBR      DIFF,J1,J2          ;SUBTRACT J2 FROM J1
          JMPT      DIFF,SKIP           ;IF NEG, CLEAR AND JMP
          AND       N,X,#00             ;CLEAR OUTPUT
          OR        N,DIFF,#00          ;LOAD OUTPUT
SKIP:
```

INT

DESCRIPTION: Integerization of a real floating point argument.

EXECUTION TIME (WORST CASE): 200nS
MEMORY WORDS REQUIRED: 5
INPUTS: X
OUTPUTS: N
CODE:

```
          STORE FPU OPT  X,              ;LET FPU CONVERT TO INT.
          STORE FPU INST ,,R/-/-
          STORE FPU INST -/INT(T)/-
          STORE FPU INST -/-/F
          LOAD FPU RES   N,F             ;READ RESULT
```

INTEG

DESCRIPTION: Performs an integration of a state variable using one of several integration routines. A fourth order Runge-Kutta method and a parallel predictor-corrector method will be shown. The parallel predictor-corrector method will be used to demonstrate improvements in execution speed resulting from parallel algorithms, and the Runge-Kutta method will show how a traditionally sequential technique can be improved with a parallel processing architecture (as well as providing starting values for the predictor-corrector method). The coefficients (K1-K4) required in the Runge-Kutta integration method will be computed in parallel for all state variables causing a system with N equations to execute in approximately the same amount of time as a sequential system with one equation.

The integration will be programmed to execute in real time, up to a maximum calculation interval. The routine will use the real-time clock values as the time variable and will update the state variables every "h" seconds. For example, if h = .01 the routine will calculate a new value of X every 10 milliseconds.

RUNGE-KUTTA INTEGRATION METHOD:

DESCRIPTION: For a program with N integrations, one integration will be allocated to a cluster of processing elements (PEs). The cluster will be responsible for calculating the coefficients for its state variable and evaluating the derivative function as necessary. If the derivative function is sufficiently complex, the allocater may divide the function among one or more PEs in the cluster to improve execution speed. The general case for computing the integration is as follows:

```
Given:    X' = F(t,x,y,...,z), X(0)=C
find:     X(i+1) = X(i) + K  where
     K = 1/6 * (K1 + 2K2 + 2K3 + K4),
     K1 = h*F(t(i),x(i),y(i), ... ,z(i)),
     K2 = h*F(t(i)+.5h, x(i)+.5K1, y(i)+.5J1, ... , z(i)+.5M1),
     K3 = h*F(t(i)+.5h, x(i)+.5K2, y(i)+.5J2, ... , z(i)+.5M2),
     K4 = h*F(t(i)+h, x(i)+K3, y(i)+J3, ... , z(i)+M3).
```

The coefficients Jn, ... ,Mn will be computed in parallel by the cluster assigned to that particular integration. This would normally be done in a sequential manner thus making the execution time proportional to the number of simultaneous equations being integrated in the system.

EXECUTION TIME (WORST CASE): 3.28uS + 4*(derivative function evaluation time)
MEMORY WORDS REQUIRED: 43 + 4*(derivative function expression)
INPUTS: X, Y, ... , Z (STATE VARIABLES)
OUTPUTS: X (INTEGRATED VARIABLE)
CODE:

```
;FPU REGISTER ASSIGNMENTS:
;RF0 - TEMP. WORKSPACE
;RF5 - CURRENT VALUE OF X (STATE VARIABLE)
;RF6 - ACCUMULATION OF K
;RF7 - H (STEP SIZE)

;
HERE1:    JMPF      OPER,HERE1          ;WAIT FOR OPERANDS
                                        ;
;EVALUATE THE DERIVATIVE FUNCTION
          MACRO     RF0=FUNCT(T,X,Y,..,Z)
;
          STORE FPU INST -/P*Q/-        ;DERIV * H
;STORE RESULT IN ACCUMULATION REG. AND RF0
          STORE FPU INST RF0,.5,/-/RF0,RF6
          STORE FPU INST -/P*Q/-        ;DIVIDE K1/2
          STORE FPU INST RF5,,RF0/-/RF0 ;STORE K1/2 IN RF0
          STORE FPU INST -/P+T/-        ;CALCU. X(i) + .5*K1
          STORE FPU INST .5,RF7,R/-/F   ;STORE RESULT
          LOAD FPU RES   TEMP,F         ;READ RESULT
;
;SEND X(i)+.5K1 TO I/O PROCESSOR FOR TRANSMISSION TO OTHER PEs IN
;SYSTEM.
          STORE     IOP,TEMP
```

```
;
;
HERE2:    JMPF      OPER,HERE2          ;WAIT FOR OPERANDS
;
;EVALUATE DERIVATIVE FUNCTION
          MACRO     RF0=FUNCT(T, X+.5K1, Y+.5J1,...,Z+.5M1)
;
          STORE FPU INST -/P*Q/-        ;COMPUTE K2 = DERIV.*H
          STORE FPU INST RF0,2,RF6/-/RF0; K2 = DERIV*H IN RF0
          STORE FPU INST -/P*Q+T/-      ; K2*2 + ACC
          STORE FPU INST -/P*Q+T/-
          STORE FPU INST RF0,.5,/-/RF6  ;STORE NEW ACCUM VALUE
          STORE FPU INST -/P*Q/-        ;DIVIDE K2/2
          STORE FPU INST RF5,,RF0/-/RF0 ;STORE K2/2
          STORE FPU INST -/P+T/-        ;CALCU. X(i) + .5*K2
          STORE FPU INST -/-/F          ;STORE RESULT
          LOAD FPU RES   TEMP,F         ;READ RESULT
          STORE     IOP,TEMP            ;SEND X(i)+.5K2 TO I/O
;PROCESSOR FOR TRANSMISSION TO OTHER PEs IN SYSTEM.
;
HERE3:    JMPF      OPER,HERE3          ;WAIT FOR OPERANDS
;
;EVALUATE NEW DERIVATIVE VALUE
          MACRO     RF0=FUNCT(T, X+.5K2, Y+.5J2,...,Z+.5M2)
;
          STORE FPU INST -/P*Q/-        ;COMPUTE K3 = DERIV.*H
          STORE FPU INST RF0,2,RF6/-/RF0; K3 = DERIV*H IN RF0
          STORE FPU INST -/P*Q+T/-      ; K3*2 + ACC
          STORE FPU INST -/P*Q+T/-
          STORE FPU INST RF5,,RF0/-/RF6 ;STORE NEW ACCUMULATOR
          STORE FPU INST -/P+T/-        ;CALCU. X(i) + K3
          STORE FPU INST RF7,,R/-/F     ;STORE RESULT
          LOAD FPU RES   TEMP,F         ;READ RESULT
          STORE     IOP,TEMP            ;SEND X(i)+K3 TO I/O
;PROCESSOR FOR TRANSMISSION TO OTHER PEs IN SYSTEM.
;
HERE4:    JMPF      OPER,HERE4          ;WAIT FOR NEW OPERANDS
;
;
;EVALUATE DERIVATIVE FUNCTION
          MACRO     RF0=FUNCT(T,X+K3,Y+J3,...,Z+M3)
          STORE FPU INST -/P*Q/-        ;K4 = DERIV*H
          STORE FPU INST RF0,,RF6/-/RF0 ;ACCUMULATE K4
          STORE FPU INST -/P+T/-        ;ACCUMULATE
          STORE FPU INST R,RF6,/-/RF6   ;K IS ALMOST COMPLETE!
          STORE FPU OP   (1/6),         ;DIVIDE K BY 6
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF6,,RF5/-/RF6 ;K IS IN RF6
          STORE FPU INST -/P+T/-        ;CALCU. X(i) + K
          STORE FPU INST -/-/RF5        ;STORE X(i+1)
; NEW STATE VARIABLE VALUE IS IN RF5
          LOAD FPU RES   TEMP,F         ;READ NEW STATE VARIABLE
          STORE     IOP,TEMP            ;SEND X(i+1) TO I/O
;PROCESSOR FOR TRANSMISSION TO OTHER PEs IN SYSTEM.
```

PARALLEL PREDICTOR-CORRECTOR METHOD:

DESCRIPTION: A parallel form of the classic predictor-corrector method for solving differential equations will be programmed using the following equations:

$$Xp(n+1) = Xc(n-1) + 2*h*F(T(n), Xp(n),...,Zp(n)), \text{ and}$$
$$Xc(n) = Xc(n-1) + h*.5*(F(T(n), Xp(n),...,Z(n)) + F(T(n-1),Xc(n-1),...,Zc(n-1)))$$

where Xc is the corrected value and Xp is the predicted value.

Using this form allows the prediction of the n+1 value while correcting the n value. Both the prediction and correction can be done concurrently. Two PEs will be employed in solving the equations, one for the predictor and one for the corrector. As in the Runge-Kutta method, one integration will be allocated to a cluster of PEs thus allowing complex functions to be evaluated with a high degree of intra-cluster processor communication with out degrading the overall system communication.

Notice that the term F(T(n),Xp(n),...,Zp(n)) is present in both the predictor and the corrector equations. If the derivative is relatively simple, the corrector PE simply re-computes the derivative function; otherwise, the derivative function computed by the predictor PE is sent to the corrector PE for use in its equation. This method would allow high efficiency since the corrector still must compute the derivative at n-1 using predicted values; therefore, the corrector could compute the derivative at n-1 while the predictor computes the derivative at n using the predicted values making the only inefficiency present the communication delay time for the transfer of Fp(n).

PREDICTOR PROGRAM:

EXECUTION TIME (WORST CASE): 960nS + function evaluation time
MEMORY WORDS REQUIRED: 14 + derivative function
INPUTS: X,Y,...,Z (STATE VARIABLES)
OUTPUTS: XPN+1
PARAMETERS:
XCN-1 - CORRECTED VALUE OF X AT TIME N-1
XPN - PREDICTED VALUE OF X AT TIME N
XPN+1 - PREDICTED VALUE OF X AT TIME N+1
CODE:

```
;FPU REGISTER ASSIGNMENT:
;RF0 - SCRATCH PAD
;RF7 - H


HERE1:    JMPF      OPER,HERE1          ;WAIT FOR OPERANDS
;
;EVALUATE THE DERIVATIVE AT N.
          MACRO     RF0=FUNCT(T,XPN,...,ZPN)
;
          LOAD FPU RES   TEMP,F
```

```
          STORE     IOP,TEMP              ;SEND Fp(n) TO CORRECTOR
;
          STORE FPU INST RF0,RF7,/-/-     ;DERIV*H
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF0,2,/-/RF0     ;STORE RESULT IN RF0
          STORE FPU INST -/P*Q/-          ;[DERIV*H]*2
          STORE FPU INST RF0,,R/-/RF0     ;STORE RESULT IN RF0
          STORE FPU OP   XCN-1,           ;ADD OLD CORRECTED
;VALUE TO NEW PREDICTED DERIVATIVE.
          STORE FPU INST -/P+T/-          ;
          STORE FPU INST -/-/F
          LOAD FPU RES   XPN+1,F          ;READ NEW PREDICTED VALUE
;
;SEND VALUE TO OTHER PEs FOR USE IN THEIR CALCULATIONS.
          STORE     IOP,XPN+1
;
          OR        XPN,XPN+1,#00         ;UPDATE Xp(n) VALUE
;
```

CORRECTOR PROGRAM:

EXECUTION TIME (WORST CASE): 1.16uS + function evaluation time + communication delay.
MEMORY WORDS REQUIRED: 16 + derivative function
INPUTS: X,Y,...,Z (STATE VARIABLES)
OUTPUTS: XCN - CORRECTED VALUE OF X AT TIME N.
PARAMETERS:
XCN-1 - CORRECTED VALUE OF X AT TIME N-1
XCN - CORRECTED VALUE OF X AT TIME N
XPN - PREDICTED VALUE OF X AT TIME N
CODE:

```
;FPU REGISTER ASSIGNMENT:
;RF0 - SCRATCH PAD
;RF7 - H (STEP INTERVAL)
;
HERE1:    JMPF      OPER,HERE1          ;WAIT FOR OPERANDS
;
;EVALUATE THE DERIVATIVE FUNCTION WITH CORRECTED VALUES AT N-1
          MACRO     RF0=FUNCT(TN-1,XCN-1,...,ZCN-1)
;
;WAIT FOR Fp(n) FROM THE PREDICTOR PE
HERE2:    JMPF      FNPSTATUS,HERE2
;
          STORE FPU OPT  FPN,
          STORE FPU INST RF0,RPN,/-/-   ;ADD TWO FUNCTIONS
          STORE FPU INST -/P+T/-
          STORE FPU INST RF7,RF0,/-/RF0 ;STORE RESULT IN RF0
          STORE FPU INST -/P*Q/-        ;RF0*H
          STORE FPU INST RF0,.5,/-/RF0
          STORE FPU INST -/P*Q/-        ;RF0*.5
          STORE FPU INST RF0,,R/-/RF0   ;PUT RESULT IN RF0
          STORE FPU OP   XCN-1,         ;STORE OLD CORRECTED VAL
```

```
          STORE FPU INST -/P+T/-          ;ADD OLD X TO RF0
          STORE FPU INST -/-/F
          LOAD FPU RES   XCN,F            ;READ NEW X VALUE
;
;SEND TO OTHER PEs FOR USE IN NEXT CALCULATION INTERVAL
          STORE     IOP,XCN
;
          OR        XCN-1,XCN,#00         ;UPDATE OLD X VALUE
;
```

ISIGN

DESCRIPTION: Append a sign (ISIGN(J1,J2)) when J1 and J2 are integers. Result is sign of J2 times absolute value of J1.

EXECUTION TIME (WORST CASE): 200nS
MEMORY WORDS REQUIRED: 5
INPUTS: J1,J2
OUTPUTS: N
CODE:

```
        STORE FPU OPT  J1,J2          ;FPU PERFORMS THIS
        STORE FPU INST R,,S/-/-       ;EXACT OPERATION.
        STORE FPU INST -/ISIGN(T)*IABS(P)/-
        STORE FPU INST -/-/F
        LOAD FPU RES   N,F            ;READ RESULT
```

LIMINT

DESCRIPTION: Limit the integrator by holding its derivative at zero while the sign of the derivative tries to drive the integrater further into the limited range. The derivative is released as soon as its sign changes to the proper direction.

EXECUTION TIME (WORST CASE): 640nS
MEMORY WORDS REQUIRED: 16
INPUTS: Y - INTEGRATOR
OUTPUTS: YD - DERIVATIVE
PARAMETERS:
IC - INITIAL CONDITION ON Y
UL - UPPER LIMIT ON Y
LL - LOWER LIMIT ON Y
CODE:

[INSERT AT THE BEGINNING OF INTEGRATION ROUTINES]

```
          STORE FPU OPT  UL,Y            ;COMPUTE UL - Y
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/P-T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   DIFF,F
          JMPF      DIFF,OK              ;IF UL-Y POS, JUMP
          NOP
          AND       YD,X,#00             ;CLEAR DERIVATIVE
OK:       STORE FPU OPT  Y,LL
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/P-T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   DIFF,F          ;READ Y - LL
          JMPF      DIFF,OK1             ;IF Y-LL POS, JMP
          NOP
          AND       YD,X,#00             ;CLEAR DERIVATIVE
OK1:
```

LSW

DESCRIPTION: The logical switch function, LSW(P,J1,J2) is implemented as follows:

```
     if  P  is true, then  N = J1,
     if  P  is false, then N = J2.
```

EXECUTION TIME (WORST CASE): 120nS
MEMORY WORDS REQUIRED: 3
INPUTS: P
OUTPUTS: N
PARAMETERS: J1, J2
CODE:

```
          JMPT      P, END              ;IF P TRUE, N=J1
          OR        N,J1,#00            ;IF P FALSE, N=J2
          OR        N,J2,#00
END:
```

MAXO

DESCRIPTION: Determine the maximum argument where the inputs are integers and the output is an integer value.

EXECUTION TIME (WORST CASE): (7*CNT + 15)*40nS
MEMORY WORDS REQUIRED: 16
INPUTS: J1, J2, J3, ... Jn
OUTPUTS: N
PARAMETERS:
CNT = NUMBER OF OPERANDS - 1.
IPA = POINTING TO BEGINNING OF STRING
(ASSUME VARIABLES ARE IN GENERAL PURPOSE REGISTERS)
CODE:

```
          OR        N, IPA, 0
AGAIN:    CPLE      COND, IPA, N        ;COMPARE CURRENT VALUE
          JMPT      COND, SKIP          ;TO CURRENT MAX.
          MFSR      COND,IPAREG
          OR        N, IPA, 0           ;IF GREATER, REPLACE OLD.
SKIP:     ADD       COND,COND,#01       ;INCREMENT IPA TO POINT
          JMPFDEC   CNT, AGAIN
          MTSR      IPAREG,COND         ;AT NEXT VALUE.
;
          STORE FPU INST RF0,,R/-/RF0
;
;WAIT FOR RESULTS FROM OTHER PE
HERE:     JMPF      OPER,HERE
          NOP
;
          STORE FPU OP   X,N
          STORE FPU INST -/MAX P,T/-
          STORE FPU INST -/-/F
          LOAD FPU INST  Y,F
          STORE     IOP,Y               ;SEND RESULT TO NEXT PE
;
```

MAX1

DESCRIPTION: Return the maximum argument where the inputs are floating point values and the output is an integer value.

EXECUTION TIME (WORST CASE): (10*CNT + 29)*40nS
MEMORY WORDS REQUIRED: 23
INPUTS: X1, X2, X3, ... Xn
OUTPUTS: N
PARAMETERS:
CNT = NUMBER OF OPERANDS - 1
IPA = POINTS TO START OF STRING
(ASSUME ALL OPERANDS ARE IN THE GENERAL PURPOSE REGISTERS)
CODE:

```
          OR          Y, IPA, 0            ;INITIALIZE FPU ACCUM.
          STORE FPU OPT  Y,
          STORE FPU INST R,,/-/-
          STORE FPU INST -/P/-
          STORE FPU INST RF0,,R/-/RF0
;
AGAIN:    STORE FPU OP   IPA               ;LET FPU FIND MAXIMUM
          STORE FPU INST -/MAX P,T/-
          STORE FPU INST RF0,,R/-/RF0      ;STORE NEW MAXIMUM
          MFSR        COND,IPAREG
          ADD         COND,COND,#01
          JMPFDEC     CNT, AGAIN
          MTSR        IPAREG,COND          ;INCREMENT IPA
;
          STORE FPU OPT  Y,                ;CONVERT MAX TO INTEGER
          STORE FPU INST ,,R/-/-
          STORE FPU INST -/INT(T)/-
          STORE FPU INST RF0,,R/-/RF0
;
;WAIT FOR RESULTS FROM OTHER PE
HERE:     JMPF        OPER,HERE
          NOP
;
          STORE FPU OP   X
          STORE FPU INST -/MAX P,T/-
          STORE FPU INST -/-/F
          LOAD FPU INST  Y,F
          STORE       IOP,Y                ;SEND RESULT TO NEXT PE
;
```

MIN0

DESCRIPTION: Determine the minimum argument where the inputs are integers and the output is an integer value.

EXECUTION TIME (WORST CASE): (7*CNT + 15)*40nS
MEMORY WORDS REQUIRED: 16
INPUTS: J1, J2, J3, ... Jn
OUTPUTS: N
PARAMETERS:
CNT = NUMBER OF OPERANDS - 1.
IPA = POINTING TO BEGINNING OF STRING
(ASSUME VARIABLES ARE IN GENERAL PURPOSE REGISTERS)
CODE:

```
          OR        N, IPA, 0
AGAIN:    CPGE      COND, IPA, N        ;COMPARE CURRENT MIN TO
          JMPT      COND, SKIP          ;CURRENT VALUE
          MFSR      COND,IPAREG
          OR        N, IPA, 0
SKIP:     ADD       COND,COND,#01       ;INCREMENT IPA TO POINT
          JMPFDEC   CNT, AGAIN
          MTSR      IPAREG,COND         ;TO NEXT VALUE
;
          STORE FPU INST RF0,,R/-/RF0
;
;WAIT FOR RESULTS FROM OTHER PE
HERE:     JMPF      OPER,HERE
          NOP
;
          STORE FPU OP   X,N
          STORE FPU INST -/MIN P,T/-
          STORE FPU INST -/-/F
          LOAD FPU INST  Y,F
          STORE     IOP,Y             ;SEND RESULT TO NEXT PE
;
```

MIN1

DESCRIPTION: Return the minimum argument where the inputs are floating point values and the output is an integer value.

EXECUTION TIME (WORST CASE): (10*CNT + 29)*40nS
MEMORY WORDS REQUIRED: 23
INPUTS: X1, X2, X3, ... Xn
OUTPUTS: N
PARAMETERS:
CNT = NUMBER OF OPERANDS - 1
IPA = POINTS TO START OF STRING
(ASSUME ALL OPERANDS ARE IN THE GENERAL PURPOSE REGISTERS)
CODE:

```
          OR        Y, IPA, 0           ;INITIALIZE FPU ACCUM.
          STORE FPU OPT  Y,
          STORE FPU INST R,,/-/-
          STORE FPU INST -/P/-
          STORE FPU INST RF0,,R/-/RF0
;
AGAIN:    STORE FPU OP   IPA            ;LET FPU FIND MINIMUM
          STORE FPU INST -/MIN P,T/-
          STORE FPU INST RF0,,R/-/RF0   ;STORE NEW MINIMUM
          MFSR      COND,IPAREG
          ADD       COND,COND,#01
          JMPFDEC   CNT, AGAIN
          MTSR      IPAREG,COND         ;POINT IPA TO NEXT VALUE
;
          STORE FPU OPT  Y,             ;CONVERT MIN TO INTEGER.
          STORE FPU INST ,,R/-/-
          STORE FPU INST -/INT(T)/-
          STORE FPU INST RF0,,R/-/RF0
;WAIT FOR RESULTS FROM OTHER PE
HERE:     JMPF      OPER,HERE
          NOP
          STORE FPU OP   X
          STORE FPU INST -/MIN P,T/-
          STORE FPU INST -/-/F
          LOAD FPU INST  Y,F
          STORE     IOP,Y               ;SEND RESULT TO NEXT PE
```

MOD

DESCRIPTION: Returns the remainder of a division (J1/J2) when the operands are integers.

EXECUTION TIME (WORST CASE): 2.2uS
MEMORY WORDS REQUIRED: 55
INPUTS: J1, J2
OUTPUTS: N
CODE:

```
;COMPUTE J1/J2 WITH REMAINDER.
          MACRO     IDIV(QUOTIENT,N,J1,J2)
;REMAINDER IS IN REGISTER N.
```

MODINT

DESCRIPTION: Provides an integration function that has a HOLD and RESET mode.

INPUTS: YD - DERIVATIVE
OUTPUTS: Y - INTEGRATOR
PARAMETERS:
IC - INITIAL CONDITION ON Y
L1, L2 - LOGICAL VARIABLES DENOTING THE MODE AS SHOWN BELOW:

L1	L2	MODE
T	F	RESET
F	F	OPERATE
T	T	OPERATE
F	T	HOLD

CODE:

```
          XOR       TEST,L1,L2
          JMPT      TEST,OPERATE        ;IF L1=L2, OPERATE
          NOP
;
          JMPT      L1,RESET            ;IF L1 TRUE, RESET
          NOP
;
;MUST BE HOLD, SO SKIP INTEGRATION
          JMP       END
          NOP
;
RESET:    OR        Y,IC,#00            ;RESET FUNCTION TO I.C.
          JMP       END
          NOP
;
OPERATE:  MACRO     INTEG(Y, IC)        ;INSERT INTEGRATION
;
END:
```

RAMP

DESCRIPTION: Generates a unity ramp function starting after time TZ and given by the function:

```
      Y = 0          T<TZ,
      Y = T-TZ       T>TZ.
```

EXECUTION TIME (WORST CASE): 400nS
MEMORY WORDS REQUIRED: 10
INPUTS: NONE
OUTPUTS: Y
CODE:

```
          STORE FPU OPT  T,TZ
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   COMP,FLAG       ;READ FPU FLAGS
          AND        COMP,COMP,08H       ;LOOK AT < FLAG
          CPEQ       COMP,COMP,08H
          JMPT       COMP,END            ;IF T<TZ QUIT + CLR
          AND        Y,Y,00
;NOW, MAKE Y = T-TZ
          LOAD FPU RES   Y,F             ;READ DIFFERENCE
END:
```

RSW

DESCRIPTION: The real switch function, LSW(P,X1,X2) is implemented as follows:

```
     if  P  is true, then  Y = X1,
     if  P  is false, then Y = X2.
```

EXECUTION TIME (WORST CASE): 120nS
MEMORY WORDS REQUIRED: 3
INPUTS: P
OUTPUTS: Y
PARAMETERS: X1, X2
CODE:

```
          JMPT      P, END              ;IF P TRUE, Y=X1
          OR        Y,X1,#00            ;IF P FALSE, Y=X2
          OR        Y,X2,#00
END:
```

RTP

DESCRIPTION: Converts a complex variable in rectangular from to a complex variable in polar form.

EXECUTION TIME (WORST CASE): 14.12uS
MEMORY WORDS REQUIRED: 164
INPUTS: X,Y
OUTPUTS: MAG, ANG
CODE:

```
          STORE FPU OPT  X,
          STORE FPU INST R,R,/-/-
          STORE FPU INST -/P*Q/-
          STORE FPU INST R,R,/-/RF0      ;PUT X SQUARED IN RF0
          STORE FPU OP   Y
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF0,,RF1/-/RF1 ;PUT Y SQUARED IN RF1
          STORE FPU INST -/P+T/-
          STORE FPU INST -/-/RF2         ;X*X + Y*Y IN RF2
;
          MACRO     F=SQRT(RF2)
;
          LOAD FPU RES   MAG,F           ;READ MAGNITUDE VALUE
;
          MACRO     RF0=FDIV(Y,X)
;
          MACRO     F=ATAN(RF0)
;
          LOAD FPU RES   ANG,F           ;READ ANGLE VALUE
```

SIGN

DESCRIPTION: Append a sign where the result is the sign of X2 times the absolute value of X1.

EXECUTION TIME (WORST CASE): 200nS
MEMORY WORDS REQUIRED: 5
INPUTS: X1,X2
OUTPUTS: Y
CODE:

```
        STORE FPU OPT  X1,X2
        STORE FPU INST R,,S/-/-       ;THE FPU PERFORMS THIS OPER.
        STORE FPU INST -/SIGN(T)*ABS(P)/-
        STORE FPU INST -/-/F
        LOAD FPU RES   Y,F            ;READ THE ANSWER
```

SIN

DESCRIPTION: Returns the sine of a real argument which must be in radians. Result will be between -1.0 and 1.0.

EXECUTION TIME (WORST CASE): 1.28uS
MEMORY WORDS REQUIRED: 32
INPUTS: X
OUTPUTS: Y
CODE:

```
;IMPLEMENT THE FOLLOWING SERIES:
;SIN(X)=X(1+Y(A1+Y(A2+Y(A3+Y(A4+Y(A5+Y(A6))))))), WHERE Y = X*X.
;
          STORE FPU OPT  X,
          STORE FPU INST R,,/-/-
          STORE FPU INST -/P/-
          STORE FPU INST RF0,RF0,/-/RF0 ;X IN RF0
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF1,S,R/-/RF1  ;X SQUARED IN RF1
          STORE FPU OP   A5,A6
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2;ACCUMULATE IN RF2
          STORE FPU OP   A4,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A3
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,1/-/RF2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF2,/-/RF2
          STORE FPU INST -/P*Q/-        ;ACCUM * X
          STORE FPU INST -/-/F
          LOAD FPU RES   Y,F            ;READ ANSWER
```

SQRT

DESCRIPTION: Computes the square root of X with a recursive routine. This routine is represented by:

X(I+1) = 0.5*[X(I) + B/X(I)], where X is an approximation of B's square root.

EXECUTION TIME (WORST CASE): 9.8uS
MEMORY WORDS REQUIRED: 31
INPUTS: B
OUTPUTS: Y
PARAMETERS:
COUNT - NUMBER OF DESIRED ITERATIONS
CODE:

```
;GET SEED FOR 1st ITERATION
          STORE     TABLE,B         ;PLACE BE ON HARDWARE LOOK-UP TABLE
          LOAD      X,TABLE         ;RETRIEVE SEED VALUE
;
;COMPUTE FIRST ITERATION
          STORE FPU OPT  X,B        ;LOAD FPU
          STORE FPU INST R,,/-/-
          STORE FPU INST S,,/P/-    ;STORE X IN RF0, B IN RF1
          STORE FPU INST -/P/RF0
          STORE FPU INST -/-/RF1
;
AGAIN:    MACRO     RF2=RECIP(X)    ;COMPUTE RECIPROCAL OF X
;
          STORE FPU INST RF2,RF1,RF0/-/-
          STORE FPU INST -/P*Q+T/-
          STORE FPU INST -/P*Q+T/-       ;CALCULATE [X + B/X]
          STORE FPU INST RF0,0.5,/-/RF0 ;STORE IN RF0
          STORE FPU INST -/P*Q/-         ;CALCULATE 0.5*[X+B/X]
          JMPFDEC   COUNT,AGAIN
          STORE FPU INST -/-/RF0         ;STORE NEW X IN RF0
;
;IF NOT ALL REQUIRED ITERATIONS HAVE BEEN DONE, DO ANOTHER.
;APPROXIMATELY 7 ;ITERATIONS WILL BE REQUIRED FOR SINGLE
;PRECISION VALUES.
```

STEP

DESCRIPTION: The STEP function outputs a zero if t < tz, and outputs an one if t > tz.

EXECUTION TIME (WORST CASE): 400nS
MEMORY WORDS REQUIRED: 10
INPUTS: NONE
OUTPUTS: Y
PARAMETERS: TZ - STARTING TIME
CODE:

```
          STORE FPU OPT  T,TZ
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/COMPARE P,T/-
          STORE FPU INST -/-/F
          LOAD FPU RES   COMP,FLAG       ;READ FPU FLAGS
          AND        COMP,COMP,08H       ;LOOK AT < FLAG
          CPEQ       COMP,COMP,08H
          JMPF       COMP,END            ;IF T>TZ TURN ON
          OR         Y,ONE,#00           ;TURN OUTPUT ON
          AND        Y,Y,00              ;MAKE OUTPUT OFF
END:
```

TAN

DESCRIPTION: Returns the tangent of an angle represented in radians.

EXECUTION TIME (WORST CASE): 1.28uS
MEMORY WORDS REQUIRED: 32
INPUTS: X
OUTPUTS: Y
CODE:

```
;IMPLEMENT THE FOLLOWING SERIES:
;TAN(X)=X(1+Y(A1+Y(A2+Y(A3+Y(A4+Y(A5+Y(A6)))))))), WHERE Y = X*X.
;
          STORE FPU OPT  X,
          STORE FPU INST R,,/-/-
          STORE FPU INST -/P/-
          STORE FPU INST RF0,RF0,/-/RF0 ;X IN RF0
          STORE FPU INST -/P*Q/-
          STORE FPU INST RF1,S,R/-/RF1  ;X SQUARED IN RF1
          STORE FPU OP   A5,A6
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2;ACCUMULATE IN RF2
          STORE FPU OP   A4,
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A3
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,R/-/RF2
          STORE FPU OP   A1
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF1,RF2,1/-/RF2
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST -/T+P*Q/-
          STORE FPU INST RF0,RF2,/-/RF2
          STORE FPU INST -/P*Q/-        ;ACCUM * X
          STORE FPU INST -/-/F
          LOAD FPU RES   Y,F            ;READ ANSWER
```

UNIF

DESCRIPTION: Used to generate a uniform random number sequence where Y is a random variable distributed between L and U.

EXECUTION TIME (WORST CASE): 1.16uS
MEMORY WORDS REQUIRED: 17
INPUTS: NONE
OUTPUTS: Y
PARAMETERS:
L - LOWER LIMIT
U - UPPER LIMIT
CODE:

```
; Y = L + (U-L)*N  WHERE N IS A RANDOM NUMBER FROM 0 TO 1.
;

          LOAD       N,RDNPTR              ;READ NEW RANDOM NUMBER
          ADD        RDNPTR,RDNPTR,#01     ;POINT TO NEW R.N.
          CPEQ       COND,RDNPTR,MAXPTR    ;SEE IF AT END
          JMPF       COND,SKIP
          NOP
          OR         RDNPTR,START,#00      ;RESET OUTPUT POINTER
SKIP:     STORE FPU OPT  U,L               ;COMPUTE U-L
          STORE FPU INST R,,S/-/-
          STORE FPU INST -/P-T/-
          STORE FPU INST R,RF0,/-/RF0      ;STORE U-L IN RF0
          STORE FPU OP   N                 ;STORE RANDOM NUMBER
          STORE FPU INST -/P*Q/-           ;COMPUTE (U-L)*N
          STORE FPU INST R,,RF0/-/RF0
          STORE FPU OP   L,
          STORE FPU INST -/P+T/-
          STORE FPU INST -/-/F             ;COMPUTE L + (U-L)*N
          LOAD FPU RES   Y,F               ;READ RESULT
```

ZHOLD

DESCRIPTION: Implements a zero order hold function in the following manner:

```
y = x  if  p is true,
y = hold  if  p is false.
```

EXECUTION TIME (WORST CASE): 120nS
MEMORY WORDS REQUIRED: 3
INPUTS: X,P
OUTPUTS: Y
CODE:

```
        JMPF      P,END              ;IF P FALSE, QUIT
        NOP
        OR        Y,X,00             ;MAKE Y = X
END:
```

FDIV (MACRO ROUTINE)

DESCRIPTION: Performs a single precision floating point division routine for 32 bit operations using a Newton-Raphson method which computes the reciprocal of the divisor and then multiplies it times the dividend to determine the quotient. This routine can be used to find the reciprocal of a value as well as performing floating point division.

EXECUTION TIME (WORST CASE): (7*ITERATIONS + 10)*40nS
EX: 1.24uS with 3 iterations
MEMORY WORDS REQUIRED: 17
INPUTS: DIVISOR, DIVIDEND
OUTPUTS: QUOTIENT(RF3), RECIPROCAL(RF0)
CODE:

```
          STORE FPU OPT  DIVISOR
          STORE FPU INST R,,/-/-
          STORE FPU INST -/P/-
          STORE FPU INST RF1,,/-/RF1          ;PUT B IN RF1
          STORE FPU INST -/RECIP_SEED/-
          STORE FPU INST RF0,RF1,2/-/RF0      ;SEED IN RF0
;READY FOR FIRST ITERATION FOR RECIPROCAL DIVISION
;EVALUATE Xi+1 = Xi*(2-b*Xi)
;
AGAIN:    STORE FPU INST -/T-P*Q/-
          STORE FPU INST -/T-P*Q/-
          STORE FPU INST -/-/RF2              ;RF2 = 2-B*X(i)
          STORE FPU INST RF0,RF2,/-/-
          STORE FPU INST -/P*Q/-
          JMPFDEC   COUNT,AGAIN               ;DO REQUIRED ITERATIONS, 3
;
          STORE FPU INST RF0,RF1,2/-/RF0
;
          STORE FPU INST R,RF0,/-/-
          STORE FPU OP   DIVIDEND             ;MULTIPLY DIVIDEND BY
                                              ;1/DIVISOR
          STORE FPU INST -/P*Q/-
          STORE FPU INST -/-/RF3              ;QUOTIENT IN RF3 AND F
```

IDIV (MACRO ROUTINE)

DESCRIPTION: Performs a signed 64 by 32 bit INTEGER division with the following parameters:

EXECUTION TIME (WORST CASE): 2.2uS
MEMORY WORDS REQUIRED: 55
INPUTS:
DIVMSW - MSW OF DIVIDEND,
DIVLSW - LSW OF DIVIDEND,
DIVISOR - 32 BIT DIVISOR
OUTPUTS:
QUOTIENT - 32 BIT QUOTIENT,
N - 32 BIT REMAINDER

```
;
          ASNE      DIVZERO,DIVISOR,00    ;CHK DIVIDE BY ZERO
          JMPF      DIVMSW,SKIP1          ;JMP IF POSITIVE
          CONST     FLAGIT,0000           ;SET FLAG 0 FOR POS.
          CPEQ      FLAGIT,FLAGIT,00      ;MAKE TRUE FOR NEG.
          SUBR      DIVLSW,DIVLSW,00      ;NEGATE L.O.WORD
          SUBRC     DIVMSW,DIVMSW,00      ;NEGATE H.O.WORD
SKIP1:    JMPF      DIVISOR,SKIP2         ;JMP IF DIVISOR POS.
          NOP
          CPEQ      FLAGIT,FLAGIT,00      ;TOGGLE FLAG
          SUBR      DIVISOR,DIVISOR,00    ;NEGATE DIVISOR
SKIP2:    MTSR      Q,DIVLSW              ;SET Q TO DIVIDEND LOW
          DIV0      N,DIVMSW              ;MAKE SHIFT AREA FOR DIV.
          DIV       N,N,DIVISOR           ;PERFORM 32 STROKE DIVISION.
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
```

```
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
          DIV       N,N,DIVISOR
;
          DIVL      N,N,DIVISOR          ;LAST STEP OF DIVIDE
          DIVREM    N,N,DIVISOR          ;REMAINDER INTO N
          MFSR      QUOTIENT,Q           ;LOAD QUOTIENT
          CPLT      OVRFLW,QUOTIENT,00   ;IF NEG, SET FLAG
          JMPF      OVRFLW,SKIP3
          CPEQ      SETMSB,SETMSB,SETMSB;SETMSB=80000000H
          CPEQ      OVRFLW,SETMSB,QUOTIENT
          CPNEQ     OVRFLW,OVRFLW,FLAGIT
SKIP3:    JMPF      POS,FLAGIT           ;NO CORRECTION, JMP
          ASEQ      DIVOVRFW,OVRFLW,00   ;IF SET OVERFLOW OCCURRED
          SUBR      QUOTIENT,QUOTIENT,00;NEGATE QUOTIENT
          SUBR      N,N,00               ;NEGATE REMAINDER
POS:
```

LIST OF REFERENCES

Beyer, W.H. 1984. CRC standard mathematical tables, 27th ed. Boca Raton: CRC Press.

Briggs, F.A., and K. Hwang. 1984. Computer architecture and parallel processing. 1984. New York:McGraw-Hill.

Cushman, Robert H. 1987. EDN's 14th annual uP/uC chip directory. EDN. 26 November. 101-187.

Gimarc, C.E. 1987. A survey of RISC processors and computers of the mid-1980s. Computer. September. 59-70.

Hannaver, G. 1986. Benchmarks for evaluation of simulation multiprocessors. West Long Branch: Electronic Associates,Inc.

Howe, C.D. 1987. How to program parallel processors. IEEE Spectrum. September. 36-41.

Hunter, C.B. 1987. Introduction to the Clipper architecture. IEEE Micro. August. 6-27.

Johnson, Mike. 1987. Am29000 user's manual. Sunnyvale: Advanced Micro Devices.

_____. 1987. System considerations in the design of the Am29000. IEEE Micro. August. 28-41.

Liniger, Werner, and Willard Miranker. 1966. Parallel methods for the numerical integration of ordinary differential equations. Mathematical Computing. vol. 21. 303-320.

Milutinovic, V.M., ed. 1988. Computer architecture concepts and systems. New York: North-Holland.

Mitchell and Gauthier, Associates, pub. 1986. The advanced continuous simulation language (ACSL) reference manual. Concord: Mitchell and Gauthier, Associates.

Ralston, Anthony, and Herbert S. Wilf. 1965. Mathematical methods for digital computers. John Wiley & Sons, Inc.

Texas Instruments, Inc. 1985. SN74AS888 SN74AS890 bit-slice processor user's guide. Dallas: Texas Instruments, Inc.

Toy, Wing, and Benjamin Zee. 1986. Computer hardware/software architecture. New Jersey: Prentice-Hall.